O'REILLY

TURING 图灵程序设计丛书

第2版



Spark 高级数据分析

Advanced Analytics with Spark, Second Edition

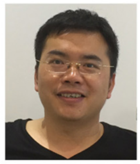
涵盖大规模数据分析中常用算法、数据集和设计模式

[美]桑迪·里扎 [美]于里·莱瑟森 [英]肖恩·欧文 [美]乔希·威尔斯 著

龚少成 邱鑫 译

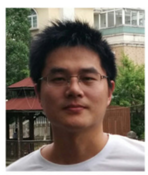
中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍



龚少成

现任万达科技集团数据工程部总经理，清华大学自动化系研究生毕业，国内专注企业级大数据平台建设的先驱者之一，曾经在Intel和Cloudera公司担任大数据技术负责人，Cloudera公司认证大数据培训讲师。



邱鑫

毕业于武汉大学，目前就职于英特尔亚太研发有限公司，是Intel大数据团队高级工程师。主要研究大数据与深度学习技术，是基于Spark的深度学习框架BigDL的核心贡献者。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

Spark高级数据分析（第2版）

Advanced Analytics with Spark, Second Edition

[美] 桑迪·里扎 [美] 于里·莱瑟森 [英] 肖恩 · 欧文 [美] 乔希 · 威尔斯 著
龚少成 邱鑫 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

Spark高级数据分析 / (美) 桑迪·里扎
(Sandy Ryza) 等著 ; 龚少成, 邱鑫 译. -- 北京 : 人
民邮电出版社, 2018.5
（图灵程序设计丛书）
ISBN 978-7-115-48252-5

Ⅰ. ①S… Ⅱ. ①桑… ②龚… ③邱… Ⅲ. ①数据处
理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2018)第076391号

内 容 提 要

本书是使用 Spark 进行大规模数据分析的实战宝典，由知名数据科学家撰写。本书在第 1 版的基础上，针对 Spark 近年来的发展，对样例代码和所使用的资料进行了大量更新。新版 Spark 使用了全新的核心 API，MLlib 和 Spark SQL 两个子项目也发生了较大变化，本书为关注 Spark 发展趋势的读者提供了与时俱进的资料，例如 Dataset 和 DataFrame 的使用，以及与 DataFrame API 高度集成的 Spark ML API。

本书适合从事数据分析的各类专业人员阅读。

◆ 著　[美] 桑迪 · 里扎　[美] 于里 · 莱瑟森
　[英] 肖恩 · 欧文　[美] 乔希 · 威尔斯
译　龚少成　邱　鑫
责任编辑　朱　巍
执行编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：15.25
字数：360千字　2018年 5 月第 1 版
印数：1 – 4 000册　2018年 5 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-2239号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

推荐序

数据的爆炸式增长和隐藏在这些数据背后的商业价值催生了一代又一代的大数据处理技术。十余年前 Hadoop 横空出世，Doug Cutting 先生将谷歌的 MapReduce 思想用开源的方式实现出来，由此拉开了基于 MapReduce 的大数据处理框架在企业中应用的序幕。近年来，Hadoop 生态系统又发展出以 Spark 为代表的新计算框架。相比 MapReduce，Spark 速度快，开发简单，并且能同时兼顾批处理和实时数据分析。Spark 起源于加州大学伯克利分校的 AMPLab，Cloudera 公司作为大数据市场上的翘楚，很早就开始将 Spark 推广到广大企业级客户并积累了大量的经验。*Advanced Analysis with Spark* 一书正是这些经验的结晶。另一方面，企业级用户在引入 Spark 技术时碰到的最大难题之一就是能够灵活应用 Spark 技术的人才匮乏。龚少成与图灵公司将 *Advanced Analysis with Spark* 翻译成中文，让国内读者第一时间用母语感受 Spark 这一新技术在数据分析和处理方面的魔力，实在是国内技术圈的幸事。能为本书作序推荐，也算是为国内企业更好地应用 Spark 技术尽自己的一份力量！

本书开篇介绍了 Spark 的基础知识，然后详细介绍了如何将 Spark 应用到各个行业。与许多图书只着重描述最终方案不同，本书作者在介绍案例时把解决问题的整个过程也展现了出来。在介绍一个主题时，并不是一开始就给出最终方案，而是先给出一个最初并不完善的方案，然后指出方案的不足，引导读者思考并逐步改进，最终得出一个相对完善的方案。这体现了工程问题的解决思路，也体现了大数据分析是一个迭代的过程。这样的论述方式更能激发读者的思考，这一点实在难能可贵。

本书英文版自第 1 版出版以来，在亚马逊网站大数据分析类图书中一直名列前茅，而且获得的多为五星级评价，可见国外读者对该书的喜爱。本书中文版译者龚少成技术扎实，在英特尔和 Cloudera 工作期间带领团队成功实施过许多国内标杆大数据平台项目，最近两年又转战万达科技集团大数据中心从零到一构建 PB 级大数据平台并支撑业务落地，而且其英语功底也相当扎实，此外我偶然得知他还是国内少数通过高级口译考试的专业人才。所以本书的中文版交给龚少成翻译实在是件让人欣慰的事情。本书中文版初稿也证实了我的判断，不仅保持了英文版的风格，而且语言也十分流畅。如果你了解 Scala 语言，还有一些统计学和机器学习基础，那么本书是你学习 Spark 时必备的图书之一！

——苗凯翔，思科中国研发公司首席技术官，前 Cloudera 公司副总裁

译者序

大数据是这几年科技和应用领域炙手可热的话题，而 Spark 又是大数据领域里最活跃的技术。随着人工智能的崛起，业内对大数据的需求不再局限于一般意义上的大数据存储、加工和分析，如何挖掘大数据的潜在价值成为新的热点。本书四位作者均在 Cloudera 公司担任过数据科学家，长期为客户提供专业的数据分析和挖掘服务。可以说，本书的出版将为 Spark 在数据分析和挖掘领域起到巨大的推动作用。

同时我们也注意到，国内介绍 Spark 数据分析方面的图书还比较匮乏，而且许多图书都停留在源代码研究的层面上。当然，这些书中也不乏非常优秀的作品，但我们认为 Spark 真正的力量在于其开发的大数据应用。所以早在本书还处于初期编写过程中时，我们就自告奋勇和作者联系中文版事宜，希望以此为中国的大数据分析事业略尽绵力。

本书在翻译过程中得到了许多人的帮助。首先要感谢我在 Cloudera 公司的前同事，也就是本书的 4 位作者。在本书的翻译过程中，由于不同语言的习惯问题，4 位作者桑迪·里扎、于里·莱瑟森、肖恩·欧文和乔希·威尔斯花了许多时间和我交流。本人之所以有幸负责本书的中文版翻译，也是承蒙肖恩·欧文的引荐。其次要感谢星环信息科技有限公司创始人孙元浩先生将我带入到大数据这个领域，让我的人生轨迹发生变化；感谢思科中国研发公司首席技术官苗凯翔博士在英特尔和 Cloudera 工作期间曾经给我的指导，让我有了端正的工作态度和价值观；感谢我的前同事田占凤博士和陈建忠的鼓励，中文版的翻译工作才得以开始。同时本书在翻译过程中还得到了 Cloudera 公司中国区前同事刘贺峰、糜君、陈飚、陈新江、李大超和张莉苹的鼎力帮助。感谢图灵公司的李松峰、岳新欣、温雪编辑在翻译过程中的指导和仔细审阅。由于本书的翻译都是在周末完成的，所以特别感谢我的妻子周幼琼在每个周末对我的照顾。

龚少成

首先非常感谢龚少成给我这次机会，使我有幸成为本书第 2 版的译者之一。

其次要感谢英特尔大数据团队的同事们，是你们带领我走进了 Spark 的时代。

最后要感谢我的妻子和孩子对我工作的理解和支持，让我腾出业余时间完成此次翻译工作。

由于译者水平有限，同时本书涉及许多课题，所以现有译文中难免存在纰漏之处。希望读者能够不吝赐教，发现问题时麻烦和译者联系。邮件请发送至 gongshaocheng@gmail.com 或 qiuxin2012cs@gmail.com。

邱鑫

序

自从在加州大学伯克利分校创立 Spark 项目起，我就时常心潮澎湃。不仅因为 Spark 可以帮助人们快速构建并行系统，更因为 Spark 帮助了越来越多的人使用大规模计算。因此看到这本介绍 Spark 高级分析的书，我非常欣慰！该书由数据科学领域 4 位专家桑迪、于里、肖恩和乔希携手打造。4 位作者研习 Spark 已久，他们在本书中跟读者分享了关于 Spark 的大量精彩内容，同时本书的案例部分同样出众！

对于这本书，我最钟爱的是它强调案例，而且这些案例都源于现实数据和实际应用。找到 1 个像样的、能在笔记本电脑上运行的大数据案例已经很难，更遑论 10 个了。但本书作者做到了！作者为大家准备好了一切，只等你在 Spark 中运行它们。更难能可贵的是，作者不仅讨论了核心算法，更倾心于数据准备和模型调优，没有这些工作，实际项目中就无法得到好的结果。认真研读此书，你应该可以吸收这些案例中的概念并直接将其运用在自己的项目中！

大数据处理无疑是当今计算领域最激动人心的方向之一，发展非常迅猛，新思想层出不穷。愿本书能帮助你在这个崭新的领域中扬帆启航！

Matei Zaharia
Databricks 公司 CTO 兼 Apache Spark 项目副总裁

前言

作者：桑迪·里扎

我不希望我的人生有很多遗憾。2011 年，某个慵懒的时刻，我正在绞尽脑汁地想如何把高难度的离散优化问题最优地分配给计算机集群处理，真是很难想到有什么好方法。我的导师跟我讲，他听说有个叫 Apache Spark 的新技术，可我基本上没当回事。Spark 的想法太好了，让人觉得有点儿不靠谱。就这样，我很快又回去接着写 MapReduce 的本科毕业论文了。时光荏苒，Spark 和我都渐渐成熟，不过令我望尘莫及的是，Spark 已然成为冉冉之星，这让人不禁感叹“Spark”（星星之火）这个双关语是多么贴切。若干年后，Spark 的价值举世皆知！

Spark 的前辈有很多，从 MPI 到 MapReduce。利用这些计算框架，我们写的程序可以充分利用大量资源，但不需要关心分布式系统的实现细节。数据处理的需求促进了这些技术框架的发展。同样，大数据领域也和这些框架关系密切，这些框架界定了大数据的范围。Spark 有望更进一步，让写分布式程序就像写普通程序一样。

Spark 能大大提升 ETL 流水作业的性能，并把 MapReduce 程序员从每天问天天不灵、问地地不应的绝望痛苦中解救出来。对我而言，Spark 的激动人心之处在于，它真正打开了复杂数据分析的大门。Spark 带来了支持迭代式计算和交互式探索的模式。利用这一开源计算框架，数据科学家终于可以在大数据集上高效地工作了。

我认为数据科学教学最有效的方法是利用实例。为此，我和同事一起编写了这本关于实际应用的书，希望它能涵盖大规模数据分析中最常用的算法、数据集和设计模式。阅读本书时不必一页一页地看，可以根据工作需要或按兴趣直接翻到相关章节。

本书内容

第 1 章结合数据科学和大数据分析的广阔背景来讨论 Spark。随后各章在介绍 Spark 数据分析时都自成一体。第 2 章通过数据清洗这一使用场景来介绍用 Spark 和 Scala 进行数据处理的基础知识。接下来几章深入讨论如何将 Spark 用于机器学习，介绍了常见应用中几个最常用的算法。其余几章则收集了一些更新颖的应用，比如通过文本隐含语义关系来查询 Wikipedia 或分析基因数据。

第2版说明

自本书第 1 版出版以来，Spark 进行了一次重大的版本更新：使用了一个全新的核心 API；MLlib 和 Spark SQL 两个子项目也发生了翻天覆地的变化。第 2 版根据新版 Spark 的最佳实践，对样例代码和所使用的资料进行了大量更新。

使用代码示例

补充材料（代码示例、练习、勘误表等）可以从 https://github.com/sryza/aas 下载[1]。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Advanced Analytics with Spark* by Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills (O'Reilly). Copyright 2015 Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills, 978-1-491-91276-8."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（前身为 Safari Books Online）是为企业、政府、教育机构和个人提供的会员制培训和参考平台。

注 1：本书中文版勘误提交及资料下载，请访问本书图灵社区页面：http://www.ituring.com.cn/book/2039。——编者注

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动教程和精选播放列表。这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

欲知更多信息，请访问 https://www.safaribooksonline.com/。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：
http://shop.oreilly.com/product/0636920056591.do

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：
https://facebook.com/oreilly

请关注我们的 Twitter 动态：
https://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：
https://www.youtube.com/oreillymedia

致谢

如果没有 Apache Spark 和 MLlib，就没有本书。所以我们应该感谢开发了 Spark 和 MLlib 并将其开源的团体，也要感谢那些添砖加瓦的数以百计的代码贡献者。

我们还要感谢本书的每一位审阅者，感谢他们花费了大量的时间来审阅本书的内容，感谢他们的专业视角，他们是 Michael Bernico、Adam Breindel、Ian Buss、Parvis Deyhim、Jeremy Freeman、Chris Fregly、Debashish Ghosh、Juliet Hougland、Jonathan Keebler、Nisha Muktewar、Frank Nothaft、Nick Pentreath、Kostas Sakellis、Tom White、Marcelo Vanzin 和另一位 Juliet Hougland。谢谢你们所有人！我们欠你们一个大人情！你们的努力大大改进了本书的结构和质量。

我（桑迪）还要感谢 Jordan Pinkus 和 Richard Wang，你们帮助我完成了风险分析章节的原理部分。

感谢 Marie Beaugureau 和 O'Reilly 出版社在本书出版和发行过程中贡献的宝贵经验和大力支持！

电子版

扫描如下二维码，即可购买本书电子版。

第 1 章

大数据分析

作者：桑迪·里扎

（数据应用）就像香肠，最好别看见它们是怎么做出来的。

——Otto von Bismarck

- 用数千个特征和数十亿个交易来构建信用卡欺诈检测模型
- 向数百万用户智能地推荐数百万产品
- 通过模拟包含数百万金融工具的投资组合来评估金融风险
- 轻松地操作成千上万个人类基因的相关数据以发现致病基因

5~10 年前想要完成上述任务几乎是不可能的。我们说生活在大数据时代，意思是指我们拥有收集、存储、处理大量信息的工具，而这些信息的规模以前我们闻所未闻。这些能力的背后是许多开源软件组成的生态系统，它们能利用大量普通计算机处理大规模数据。Apache Hadoop 之类的分布式系统已经进入主流，并被广泛部署在几乎各个领域的组织里。

但就像锉刀和石头本身并不构成雕塑一样，有了工具和数据并不等于就可以做有用的事情。这时我们就需要数据科学了。雕刻是利用工具将原始石材变成普通人都能看懂的雕塑，数据科学则是利用工具将原始数据变成对不懂数据科学的普通人有价值的东西。

通常，“做有用的事情”指给数据加上模式并用 SQL 来回答问题，比如：“注册过程中许多用户进入到第三个页面，其中有多少用户年龄超过 25 岁？”如何架构一个数据仓库，并组织信息来回答此类问题，涉及的面很广，本书不对其细节过多赘述。

有时候“产生价值”需要多付出一些努力。SQL 可能仍扮演重要角色，但为了处理数据的特质或进行复杂分析，人们需要一个更灵活、更易用的，且在机器学习和统计方面功能更丰富的编程模式。本书将重点讨论此类型的分析。

长久以来，人们利用 R、PyData 和 Octave 等开源框架可以在小数据集上进行快速分析和建模。只需不到 10 行代码，就可以利用数据集的一部分数据构建出机器学习模型，再利用该模型预测其余数据的分类。如果多写几行代码，我们还能处理缺失数据，尝试多个模型并从中找出最佳模型，或者用一个模型的结果作为输入来拟合另一个模型。但如果数据集巨大，必须利用大量计算机来达到相同效果，我们该怎样做呢？

一个可能正确的方法是简单扩展这些框架，使之能运行在多台机器上，保留框架的编程模型，同时重写其内核，使之在分布式环境下能顺利运行。但是，分布式计算难度大，我们必须重新思考在单机系统中的许多基本假设，在分布式环境下是否依然成立。比如，由于集群环境下数据需要在多个节点间切分，网络传输速度比内存访问慢几个数量级，如果算法涉及宽数据依赖，情况就很糟糕。随着机器数量的增加，发生故障的概率也相应增大。这些实际情况要求编程模式适配底层系统：编程模式要防止不当选项，并简化高度并行代码的编写。

当然，除了像 PyData 和 R 这样在软件社区里表现优异的单机工具，数据分析还用到其他工具。在科学领域，比如常常涉及大规模数据的基因学，人们使用并行计算框架已经有几十年的历史了。今天，在这些领域处理数据的人大多数都熟悉 HPC（high-performance computing，高性能计算）集群计算环境。然而，PyData 和 R 的问题在于它们很难扩展。同样，HPC 的问题在于它的抽象层次较低，难于使用。比如要并行处理一个大 DNA 测序文件，人们需要手工将该文件拆成许多小文件，并为每个小文件向集群调度器提交一个作业。如果某些作业失败，用户需要检查失败并手工重新提交。如果操作涉及整个数据集，比如对整个数据集排序，庞大的数据集必须流入单个节点，否则科学家就要用 MPI 这样底层的分布式框架。这些底层框架使用难度大，用户必须精通 C 语言和分布式 / 网络系统。

为 HPC 环境编写的工具往往很难将内存数据模型和底层存储模型独立开来。比如很多工具只能从单个流读取 POSIX 文件系统数据，很难自然并行化，不能用于读取数据库等其他后台存储。最近，Hadoop 生态系统提供了抽象，让用户使用计算机集群就像使用单台计算机一样。该抽象自动拆分文件并在多台计算机上分布式存储，自动将工作拆分成若干粒度更小的任务并分布式执行，出错时自动恢复。Hadoop 生态系统将大数据集处理涉及的许多琐碎工作自动化，并且启动开销比 HPC 小得多。

1.1 数据科学面临的挑战

数据科学界有几个硬道理是不能违背的，Cloudera 数据科学团队的一项重要职责就是宣扬这些硬道理。一个系统要想在海量数据的复杂数据分析方面取得成功，必须明白这些硬道

理，至少不能违背。

第一，成功的分析中，绝大部分工作是数据预处理。数据是混乱的，在让数据产生价值之前，必须对数据进行清洗、处理、融合、挖掘和许多其他操作。特别是大数据集，由于人们很难直接检查，为了知道需要哪些预处理步骤，甚至需要采用计算方法。一般情况下，即使在模型调优阶段，在整个数据处理管道的各个作业中，花在特征提取和选择上的时间比选择和实现算法的时间还要多。

比如，在构建网站欺诈交易检测模型时，数据科学家需要从许多可能的特征中进行选择。这些特征包括必填项、IP 地址信息、登录次数、用户浏览网站时的点击日志等。在将特征转换成适用于机器学习算法的向量时，每个特征可能都会有不同的问题。系统需要支持更灵活的转换，远远不止是将二维双精度数组转换成一个数学模型那么简单。

第二，**迭代**是数据科学的基础之一。建模和分析经常需要对一个数据集进行多次遍历。这其中一方面是由机器学习算法和统计过程本身造成的。常用的优化过程，比如随机梯度下降和最大似然估计，在收敛前都需要多次扫描输入数据。数据科学家自身的工作流程也涉及迭代。在初步调查和理解数据集时，一个查询的结果往往给下一个查询带来启示。在构建模型时，数据科学家往往很难在第一次就得到理想的结果。选择正确的特征，挑选合适的算法，运行恰当的显著性测试，找到合适的超参数，所有这些工作都需要反复试验。框架每次访问数据都要读磁盘，这样会增加时延，降低探索数据的速度，限制了数据科学家进行试验的次数。

第三，构建完表现卓越的模型不等于大功告成。数据科学的目标在于让数据对不懂数据科学的人有用。把模型以许多回归权值的形式存成文本文件，放在数据科学家的计算机里，这样做根本没有实现数据科学的目标。数据推荐引擎和实时欺诈检测系统是最常见的数据应用。这些应用中，模型作为生产服务的一部分，需要定期甚至是实时重建。

在这些场景中，有必要区别是**试验环境**下的分析还是**生产环境**下的分析。在试验环境下，数据科学家进行探索性分析。他们想理解工作数据集的本质。他们将数据图形化并用各种理论来测试。他们用各种特征做试验，用辅助数据源来增强数据。他们试验各种算法，希望从中找到一两个有效算法。在生产环境下，构建数据应用时，数据科学家进行操作式分析。他们把模型打包成服务，这些服务可以作为现实世界的决策依据。他们跟踪模型随时间的表现，哪怕是为了将模型准确率提高一个百分点，他们都会精心调整模型并且乐此不疲。他们关心服务 SLA 和在线时间。由于历史原因，探索性分析经常使用 R 之类的语言，但在构建生产应用时，数据处理过程则完全用 Java 或 C++ 重写。

当然，如果用于建模的原始代码也可用于生产应用，那就能节省每个人的时间。但像 R 之类的语言运行缓慢，很难将其与生产基础设施的技术平台进行集成，而 Java 和 C++ 之类的语言又很难用于探索性分析。它们缺乏交互式数据操作所需的 REPL（read-evaluate-print-

loop，读取 – 计算 – 打印 – 循环）环境，即使是简单的转换，也需要写大量代码。人们迫切需要一个既能轻松建模又适合生产系统的框架。

1.2 认识Apache Spark

该介绍 Apache Spark 了。Spark 是一个开源框架，作为计算引擎，它把程序分发到集群中的许多机器，同时提供了一个优雅的编程模型。Spark 源自加州大学伯克利分校的 AMPLab，现在已被捐献给了 Apache 软件基金会。可以这么说，对于数据科学家而言，真正让分布式编程进入寻常百姓家的开源软件，Spark 是第一个。

了解 Spark 的最好办法莫过于了解相比于它的前辈，即 Apache Hadoop 的 MapReduce，Spark 有哪些进步。MapReduce 革新了海量数据的计算方式，为运行在成百上千台机器上的并行程序提供了简单的编程模型。MapReduce 引擎几乎可以做到线性扩展：随着数据量的增加，可以通过增加更多的计算机来保持作业时间不变。而且 MapReduce 是健壮的。故障虽然在单台机器上很少出现，但在数千个节点的集群上却总是出现。对于这种情况，MapReduce 也能妥善处理。它将工作拆分成多个小**任务**，能优雅地处理失败的任务，并且不影响任务所属作业的正确执行。

Spark 继承了 MapReduce 的线性扩展性和容错性，同时对它做了一些重量级扩展。首先，Spark 摒弃了 MapReduce 先 map 再 reduce 这样的严格方式，Spark 引擎可以执行更通用的有向无环图（directed acyclic graph，DAG）算子。这就意味着，在 MapReduce 中需要将中间结果写入分布式文件系统时，Spark 能将中间结果直接传到流水作业线的下一步。在这方面，它类似于 Dryad（https://www.microsoft.com/en-us/research/project/dryad/）。Dryad 也是从 MapReduce 衍生出来的，起源于微软研究院。其次，它也完善了这种能力，通过提供许多转换操作，用户可以更自然地表达计算逻辑。Dryad 更加面向开发人员，其流式 API 可以做到用几行代码表示复杂的流水作业。

再次，Spark 扩展了前辈们的内存计算能力。它的 Dataset 和 DataFrame 抽象使开发人员将流水处理线上的任何点物化在跨越集群节点的内存中。这样后续步骤如果需要相同数据集就不必重新计算或从磁盘加载。这个特性使 Spark 可以应用于以前分布式处理引擎无法胜任的应用场景中。Spark 非常适用于涉及大量迭代的算法，这些算法需要多次遍历相同的数据集。Spark 也适用于反应式（reactive）应用，这些应用需要扫描大量内存数据并快速响应用户的查询。

或许最重要的是，Spark 契合了前面提到的数据科学领域的硬道理。它认识到构建数据应用的最大瓶颈不是 CPU、磁盘或者网络，而是分析人员的生产率。通过将预处理到模型评价的整个流水线整合在一个编程环境中，Spark 大大加速了开发过程。这一点尤为值得称赞。Spark 编程模型富有表达力，在 REPL 下包装了一组分析库，省去了多次往返 IDE 的开销。而这些开销对诸如 MapReduce 等框架来说是无法避免的。Spark 还避免了采样和从

Hadoop 分布式文件系统（the Hadoop distributed file system，HDFS) 来回倒腾数据所带来的问题，这些问题是 R 之类的框架经常遇到的。分析人员在数据上做实验的速度越快，他们从数据中挖掘出价值的可能性就越大。

在数据处理和 ETL 方面，Spark 的目标是成为大数据界的 Python 而不是大数据界的 MATLAB。作为一个通用的计算引擎，它的核心 API 为数据转换提供了强大的基础，它独立于统计学、机器学习或矩阵代数的任何功能。它的 Scala 和 Python API 让我们可以用表达力极强的通用编程语言编写程序，还可以访问已有的库。

Spark 的内存缓存使它适用于微观和宏观两个层面的迭代计算。机器学习算法需要多次遍历训练集，可以将训练集缓存在内存里。在对数据集进行探索和初步了解时，数据科学家可以在运行查询的时候将数据集放在内存中，也很容易将转换后的版本缓存起来，这样就节省了访问磁盘的开销。

最后，Spark 在探索型分析系统和操作型分析系统之间搭起一座桥梁。我们经常说，数据科学家比统计学家更懂软件工程，比软件工程师更懂统计学。基本上讲，Spark 比探索型系统更像操作型系统，比操作型系统中常见的技术更善于数据探索。Spark 从根本上是为性能和可靠性而生的。由于构建于 JVM 之上，它可以利用 Java 技术栈里的许多操作和调试工具。

Spark 还紧密集成 Hadoop 生态系统里的许多工具。它能读写 MapReduce 支持的所有数据格式，可以与 Hadoop 上的常用数据格式，如 Apache Avro 和 Apache Parquet（当然也包括古老的 CSV），进行交互。它能读写 NoSQL 数据库，比如 Apache HBase 和 Apache Cassandra。它的流式处理组件 Spark Streaming 能连续从 Apache Flume 和 Apache Kafka 之类的系统读取数据。它的 SQL 库 SparkSQL 能和 Apache Hive Metastore 交互，而且通过 Hive on Spark，Spark 还能替代 MapReduce 作为 Hive 的底层执行引擎。它可以运行在 Hadoop 集群调度和资源管理器 YARN 之上，这样 Spark 可以和 MapReduce 及 Apache Impala 等其他处理引擎动态共享集群资源和管理策略。

1.3 关于本书

本书接下来的部分不会讨论 Spark 的优缺点。还有其他一些话题本书也不会涉及。本书会介绍 Spark 的流式编程模型和 Scala 基础知识，但它不是 Spark 参考书或参考大全，不会讲 Spark 技术细节。它也不是机器学习、统计学、线性代数的参考书，但在讲到这些知识的时候，许多章节会提供一些背景知识。

另一方面，本书将帮助读者建立用 Spark 在大规模数据集上进行复杂分析的感觉。我们会讲述整个处理过程：不但涉及模型的构建和评价，也会讲述数据清洗、数据预处理和数据探索，并会花费笔墨描述怎样将结果变成生产应用。我们认为最好的教学方法是运用实

例，所以在快速介绍完 Spark 及其生态系统之后，本书其余各章分别讨论了在不同领域使用 Spark 进行数据分析的实例，每个实例都自成一体。

如果可能的话，我们要做的不只是提供解决方案。我们会描述数据科学的整个工作流程，包括它所有的迭代、无解以及需要重新开始的情况。本书将有助于读者熟悉 Scala、Spark、机器学习和数据分析。但这都是为了一个更大的目标服务，我们希望本书首先教会读者如何完成本章开头部分提到的任务。每一章虽然只有薄薄的 20 来页，但我们会力求把怎样构建一个此类数据应用讲清楚、讲透彻。

1.4 第2版说明

2015 年和 2016 年 Spark 变化很大，2016 年 7 月 Spark 发布了 2.0 版本。其中改变最大的是 Spark 的核心 API。在 Spark 2.0 以前的版本中，Spark 的 API 主要围绕一个可以跨节点分布的、延迟实例化对象集合的**弹性分布式数据集**（Resilient Distributed Dataset，RDD）而构建。

虽然 RDD 使用了一套强大而富有表达力的 API，但是仍然存在两个主要的问题。第一，RDD 难以高效且稳定地执行任务。由于依赖 Java 和 Python 对象，RDD 对内存的使用效率较低，而且会导致 Spark 程序受长时间垃圾回收的影响。它们还将执行计划（execution plan）与 API 捆绑到了一起，给用户优化应用程序造成了沉重的负担。例如，传统 RDBMS（关系数据库管理系统）可以根据关联表的大小来选择最优的关联策略（join strategy），而 Spark 需要用户自己来做这个选择。第二，Spark 的 API 忽视了一个事实——数据往往能用一个结构化的关系形式来表示；当出现这种情况的时候，API 应该提供一些原语，使数据更加易于操作，比如允许用户使用列的名字来访问数据，而不是通过元组中的序数位置。

Spark 2.0 用 Dataset 和 DataFrame 替换掉 RDD 来解决上述问题。Dataset 与 RDD 十分相似，不同之处在于 Dataset 可以将它们所代表的对象映射到**编码器**（encoder），从而实现了一种更为高效的内存表示方法。这就意味着 Spark 程序可以执行得更快、使用更少内存，而且执行时间更好预测。Spark 还在数据集和执行计划之间加入了一个优化器，这意味着 Spark 能对如何执行做出更加智能的决策。DataFrame 是 Dataset 的子类，专门用于存储关系型数据（也就是用行和固定列表示的数据）。为了理解列的概念，Spark 提供了一套更干净的、富有表达力的 API，同时也加入了很多性能优化。举个例子，如果 Spark 知道了仅其中一部分列会被用到，它就能避免将用不到的列载入内存中。还有许多转换操作之前需要使用用户定义函数（user-defined function，UDF）来表示，现在可以在 API 中直接调用了。这对于 Python 用户来说十分有用，因为 Spark 在内部执行这些转换操作比 Python 中定义的函数要快得多。DataFrame 还可以与 Spark SQL 互相操作，这意味着用户可以写一个 SQL 查询来获取一个 DataFrame，然后选择一种 Spark 支持的语言对这个 DataFrame 进

行编程操作。尽管新 API 与旧 API 看起来十分相似，但是很多细节发生了改变，因此几乎所有的 Spark 程序都要更新。

除了核心 API 的变化以外，Spark 2.0 还见证了机器学习 API 和统计分析 API 的巨大变化。在之前的版本中，每个机器学习算法都有一套自己的 API。如果用户想要准备算法需要的输入数据，或者将一个算法的输出提供给另外一个算法，都需要写一套它们自己的自定义编制代码。Spark 2.0 包含了 Spark ML API，它引入了一个框架，可以将多种机器学习算法和特征转换步骤管道化。这个 API 受 Python 的流行框架 Scikit-Learn API 启发，以评估器（estimator）和转换器（transformer）为中心，转换器从数据中学习参数，然后用这些参数来转换数据。Spark ML API 与 DataFrame API 高度集成，使得在关系型数据上训练机器学习模型变得更容易。例如，用户可以通过名字访问特征，而不用数组下标。

总体来说，Spark 的这些变化导致本书第 1 版中的很多内容都过时了。因此，第 2 版更新了所有的章节，并尽可能地使用最新的 API。此外，我们还删除了一些无关的章节。例如，第 1 版附录介绍了 API 的细节，第 2 版中将其删除了，一定程度上是因为现在 Spark 可以自动处理，无须用户干预。随着 Spark 进入了一个成熟而稳定的新时代，我们希望通过第 2 版的这些更新，本书在今后几年内会保持对 Spark 数据分析的参考价值。

第 2 章

用Scala和Spark进行数据分析

作者：乔希·威尔斯

世上无难事，只要肯耐烦。

——David Foster Wallace

数据清洗是数据科学项目的第一步，往往也是最重要的一步。许多灵巧的分析最后功败垂成，原因就是分析的数据存在严重的质量问题，或者数据中某些因素使分析产生偏见，或使数据科学家得出根本不存在的规律。

尽管数据清洗很重要，但数据科学相关的许多教材和课程都不讲述数据清洗，抑或一笔带过。造成这种现象的原因其实很简单：数据清洗实在是很琐碎。然而"磨刀不误砍柴工"，只有事先做了这种沉闷乏味的工作，后面你才能领略到应用机器学习算法解决新问题时的酣畅淋漓。许多道行尚浅的数据科学家往往急于求成，对数据草草处理就进行下一步工作，等到运行算法后，却发现数据有严重的质量问题（可能是计算量太大），或最后得出的结果完全不合理。

"垃圾进垃圾出"这样浅显的道理大家都明白，危害更大的是：数据看似合理却有很严重（但第一眼看不出来）的质量问题，你根据这样的数据也得到了看似合理的答案。许多数据科学家丢饭碗，往往就是因为这样错误地得出了重要结论。

数据科学家最为人称道的是在数据分析生命周期的每一个阶段都能发现有意思、有价值的问题。在一个分析项目的早期阶段，你投入的技能和思考越多，对最终的产品就越有信心。

当然，说起来容易做起来难。对于数据科学行业来说，这就像是告诉小孩子要多吃蔬菜。相比数据清洗，摆弄 Spark 之类新潮的工具，用它们构建花哨的机器学习算法，开发流式数据处理引擎和分析海量图数据，要好玩得多。如果要介绍如何用 Spark 和 Scala 进行数据处理，有没有一种比练习数据清洗更好的方法呢？

2.1 数据科学家的Scala

对数据处理和分析，数据科学家往往都自己钟爱的工具，比如 R 或者 Python。除非不得已，数据科学家常常会坚持用他们所钟爱的工具，对于手头上的工作，他们总想方设法地沿用这些工具。即使情况再顺利，想让数据科学家采用新的工具、学习新语法和新使用模式，都困难重重。

为了能在 R 或 Python 里直接用 Spark，Spark 上开发了专门的类库和工具包。Python 有个非常好用的工具包叫作 PySpark，第 11 章有几个例子介绍了它的用法。但是本书大部分例子还是用 Scala 语言编写的。Spark 框架是用 Scala 语言编写的，在向数据科学家介绍 Spark 时，采用与底层框架相同的编程语言有很多好处。

- **性能开销小**

  为了能在基于 JVM 的语言（比如 Scala）上运行用 R 或 Python 编写的算法，我们必须在不同环境中传递代码和数据，这会付出代价，而且在转换过程中信息时有丢失。但是，如果数据分析算法用 Spark Scala API 编写，你会对程序正确运行更有信心。

- **能用上最新的版本和最好的功能**

  Spark 的机器学习、流处理和图分析库全都是用 Scala 写的，而新功能对 Python 和 R 绑定支持可能要慢得多。如果想用 Spark 的全部功能（而不用花时间等待它移植到其他语言绑定），恐怕你必须学点儿 Scala 基础知识，如果想扩展这些 Spark 已有功能来解决你手头上的新问题，就更要深入了解 Scala 了。

- **有助于你了解Spark的原理**

  即使在 Python 或 R 中调用 Spark，API 仍然反映了底层计算原理，它是 Spark 从其开发语言 Scala 继承过来的。如果你知道如何在 Scala 中使用 Spark，即使你平时主要还是在其他语言中使用 Spark，你还是会更理解系统，因此会更好地“用 Spark 思考”。

学习在 Scala 中用 Spark 还有一个好处。由于 Spark 不同于其他任何一种数据分析工具，这个好处解释起来会有点儿困难。如果你曾经用过 R 或 Python 从数据库读取数据并分析，肯定经历过用一种语言（SQL）读取和操作大量存储在远程集群的数据，然后用另一种语言（Python 或 R）来操作和展现存储在你本地机器上的信息。如果想把一部分计算通过 SQL UDF 放到数据库引擎中，你需要切换到另一种编程环境（如 C++ 或者 Java），并且还要了解数据库的内部细节。如果你一直这么做，时间长了你可能都不会再想这种方式有没有问题。

使用 Spark 和 Scala 做数据分析则是一种完全不同的体验，因为你可以选择用同样的语言完成**所有事情**。借助 Spark，你用 Scala 代码读取集群上的数据。接着，你把 Scala 代码发送到集群上完成相同的转换，这些转换跟你刚刚对本地数据所做的转换完全一样，但数据却在集群上——这就是精妙之处。即便用 Spark SQL 这样的高阶语言，也可以写好内联 UDF，用 Spark SQL 引擎注册，然后使用 UDF——根本不用切换环境。

在同一个环境中完成所有数据处理和分析，不用考虑数据本身在何处存放和在何处处理，这简直妙不可言。这种感觉只有你亲身经历才体会得到。我们也想确保书中的示例能够让你感受到我们首次使用 Spark 时体验到的那种魔术般的感觉。

2.2 Spark编程模型

Spark 编程始于数据集，而数据集往往存放在分布式持久化存储之上，比如 HDFS。编写 Spark 程序通常包括一系列相关步骤。

(1) 在输入数据集上定义一组转换。
(2) 调用 action，可以将转换后的数据集保存到持久化存储上，或者把结果返回到驱动程序的本地内存。
(3) 运行本地计算，处理分布式计算的结果。本地计算有助于你确定下一步的转换和 action。

从 1.2 版本到 2.1 版本，Spark 变得成熟了，处理上述步骤的工具的数量和质量也大大提升。在完成分析任务时，你可以搭配使用复杂 SQL 查询、机器学习库以及自定义代码。Spark 社区这几年开发了各种高阶抽象，利用这些抽象，你可以花更少的时间来解决更多的问题。但是所有这些高阶抽象都是基于存储与执行的相互作用，从 Spark 诞生起就一直是这样。Spark 优美地搭配这两类抽象，可以将数据处理管道中的任何中间步骤缓在内存里以备后用。了解这些原则可以帮助你更好地利用 Spark 做数据分析。

2.3 记录关联问题

本章我们要研究的主题在许多文献和实践中被冠以许多不同的名称：身份解析、记录去重、合并－清除，以及列表清洗。想了解这个主题的方案和技术概况，我们需要参考这个主题的所有相关研究论文。但是由于不同文献和实践中同一个概念使用不同的名称，我们很难找到所有相关论文。在搞清楚数据清洗这个问题之前，我们得请数据科学家把与数据清洗这个概念相关的令人混淆的许多不同名称给去去重。这真让人觉得讽刺！为了方便本章余下部分论述，我们把这个问题称为**记录关联**（record linkage）。

问题的大概情况如下：我们有大量来自一个或多个源系统的记录，其中多种不同的记录可能代表相同的基础实体，比如客户、病人、业务地址或事件。每个实体有若干属性，比如姓名、地址、生日。我们需要根据这些属性找到那些代表相同实体的记录。不幸的

是，有些属性值有问题：格式不一致，或有笔误，或信息缺失。如果简单地对这些属性作相等性测试，就会漏掉许多重复记录。举个例子，我们看看表 2-1 列出的几家商店的记录。

表 2-1：记录关联问题的难点

名　　称	地　　址	城　　市	州	电　　话
Josh's Coffee Shop	1234 Sunset Boulevard	West Hollywood	CA	(213)-555-1212
Josh Coffee	1234 Sunset Blvd West	Hollywood	CA	555-1212
Coffee Chain #1234	1400 Sunset Blvd #2	Hollywood	CA	206-555-1212
Coffee Chain Regional Office	1400 Sunset Blvd Suite 2	Hollywood	California	206-555-1212

表中前两行其实指同一家咖啡店，但由于数据录入错误，这两项看起来是在不同城市（West Hollywood 和 Hollywood）。相反，表中后两行其实是同一家咖啡连锁店的不同业务部门，尽管它们有相同的地址：地址 1400 Sunset Blvd #2 是咖啡店的实际地址，另一个地址 1400 Sunset Blvd Suite 2 则是公司在当地的一个办公室地点。后两项给的都是公司 Seattle 总部的官方电话号码。

这个例子清楚地说明了记录关联为什么很困难：即使两组记录看起来相似，但针对每一组中的条目，我们确定它是否重复的标准不一样。这种区别我们人类很容易理解，计算机却很难了解。

2.4　小试牛刀：Spark shell和`SparkContext`

我们的样例数据集来自加州大学欧文分校机器学习资料库（UC Irvine Machine Learning Repository），这个资料库为研究和教学提供了大量非常好的数据源，这些数据源非常有意义，并且是免费的。我们要分析的数据集来源于一项记录关联研究，这项研究是德国一家医院在 2010 年完成的。这个数据集包含数百万对病人记录，每对记录都根据不同标准来匹配，比如病人姓名（名字和姓氏）、地址、生日。每个匹配字段都被赋予一个数值评分，范围为 0.0 到 1.0，分值根据字符串相似度得出。然后这些数据交由人工处理，标记出哪些代表同一个人，哪些代表不同的人。为了保护病人隐私，创建数据集的每个字段的原始值被删除了。病人的 ID、字段匹配分数、匹配对标记（包括匹配的和不匹配的）等信息是公开的，可用于记录关联研究。

首先我们从资料库中下载数据，请在命令行中输入：

```
$ mkdir linkage
$ cd linkage/
$ curl -L -o donation.zip https://bit.ly/1Aoywaq
$ unzip donation.zip
$ unzip 'block_*.zip'
```

如果手头有 Hadoop 集群，可以先在 HDFS 上为块数据创建一个目录，然后将数据集文件复制到 HDFS 上：

```
$ hadoop fs -mkdir linkage
$ hadoop fs -put block_*.csv linkage
```

本书示例和代码假定读者使用 Spark 2.1.0。可以在 Spark 项目网站（https://spark.apache.org/downloads.html）获取各个版本的 Spark 软件。想了解如何在集群或本地机器上安装 Spark 环境，请参考 Spark 官方文档。

现在准备工作就绪，可以启动 `spark-shell` 了。`spark-shell` 是 Scala 语言的一个 REPL 环境，它同时针对 Spark 做了一些扩展。如果这是你第一次见到 REPL 这个术语，可以把它看成一个类似 R 的控制台：可以在其中用 Scala 编程语言定义函数并操作数据。

如果你有一个 Hadoop 集群，并且 Hadoop 版本支持 YARN，通过为 Spark master 设定 `yarn` 参数值，就可以在集群上启动 Spark 作业：

```
$ spark-shell --master yarn --deploy-mode client
```

如果你是在自己的计算机上运行示例，可以通过设定 `local[N]` 参数来启动本地 Spark 集群，其中 `N` 代表运行的线程数，或者用 * 表示使用机器上所有可用的核数。比如，要在一个 8 核的机器上用 8 个线程启动一个本地集群，可以输入以下命令：

```
$ spark-shell --master local[*]
```

在本地环境下，书中示例同样能运行。不过，这时传入的文件路径是本地路径，而不是以 `hdfs://` 开头的 HDFS 路径。注意，还需要通过 `cp block_*.csv` 将文件复制到指定的本地目录，而不是用包含解压文件的目录，因为该目录不仅包含 .csv 文件，还包含许多其他文件。

本书其他 `spark-shell` 示例中不会出现 `--master` 参数，但根据环境通常需要设定该参数。

为了 Spark shell 能充分利用资源，可能还需要额外设定一些参数。比如，当 Spark 运行于本地 master 模式，可以用 `--driver-memory 2g`，这样就设定了一个本地进程使用 2 GB 内存。YARN 内存设置会更复杂，相关的选项（如 `--executor-memory` 等参数）设置可以参考 Spark on YARN 的官方文档（https://spark.apache.org/docs/latest/running-on-yarn.html）。

运行完上述命令后，可以看到 Spark 在初始化过程中的日志消息。与此同时，也能看到一点儿 ASCII 艺术体字样，之后又是一段日志和提示符：

```
Spark context Web UI available at http://10.0.1.39:4040
Spark context available as 'sc' (master = local[*], app id = ...).
Spark session available as 'spark'.
Welcome to
```

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

如果你是第一次用 Spark shell（或任何类似的 Scala REPL），可以运行 `:help` 命令，该命令列出了 shell 的所有命令。运行 `:history` 或 `:h?`，可以帮你找到之前在某个会话中写过，但一时又想不起来的变量或函数名称。运行 `:paste`，可以帮你插入剪贴板中的代码，这是学习本书和使用本书源代码必需的。

除了关于 `:help` 的提示，Spark 日志消息还显示“Spark context available as sc.”。`sc` 在这里是 `SparkContext` 的一个引用，它负责协调集群上 Spark 作业的执行。继续在命令行中输入 `sc`：

```
sc
...
res: org.apache.spark.SparkContext =
  org.apache.spark.SparkContext@DEADBEEF
```

REPL 会以字符形式打印对象。`SparkConext` 对象就是名字加上十六进制的对象内存地址（示例中显示的 `DEADBEEF` 是占位符，具体值每次运行时都不一样）。

`sc` 变量确实方便，但它的作用是什么呢？ `SparkContext` 是一个对象，是对象当然就有方法。想要在 Scala REPL 中查看这些方法，输入变量名加点号再加 Tab 键即可：

```
sc.[\t]
...
!=                          hashCode
##                          isInstanceOf
+                           isLocal
->                          isStopped
==                          jars
accumulable                 killExecutor
accumulableCollection       killExecutors
accumulator                 listFiles
addFile                     listJars
addJar                      longAccumulator
...  (lots of other methods)
getClass                    stop
getConf                     submitJob
getExecutorMemoryStatus     synchronized
getExecutorStorageStatus    textFile
getLocalProperty            toString
getPersistentRDDs           uiWebUrl
getPoolForName              union
```

```
getRDDStorageInfo          version
getSchedulingMode          wait
hadoopConfiguration        wholeTextFiles
hadoopFile                 →
```

`SparkContext` 有很多方法，但接下来我们使用最多的方法用于创建 RDD。RDD 是 Spark 所提供的最基本的抽象，代表分布在集群中多台机器上的对象集合。Spark 有两种方法可以创建 RDD：

- 用 `SparkContext` 基于外部数据源创建 RDD，外部数据源包括 HDFS 上的文件、通过 JDBC 访问的数据库表或 Spark shell 中创建的本地对象集合；
- 在一个或多个已有 RDD 上执行转换操作来创建 RDD，这些转换操作包括记录过滤、对具有相同键值的记录做汇总、把多个 RDD 关联在一起等。

利用 RDD 可以很方便地描述对数据要进行的一串小而独立的计算步骤。

弹性分布式数据集（RDD）

RDD 以**分区**（partition）的形式分布在集群中的多个机器上，每个分区代表了数据集的一个子集。分区定义了 Spark 中数据的并行单位。Spark 框架并行处理多个分区，一个分区内的数据对象则是顺序处理。创建 RDD 最简单的方法是在本地对象集合上调用 `SparkContext` 的 `parallelize` 方法。

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
...
rdd: org.apache.spark.rdd.RDD[Int] = ...
```

第一个参数代表待并行化的对象集合，第二个参数代表分区的个数。当要对一个分区内的对象进行计算时，Spark 从驱动程序进程里获取对象集合的一个子集。

要在分布式文件系统（比如 HDFS）上的文件或目录上创建 RDD，可以给 `textFile` 方法传入文件或目录的名称：

```
val rdd2 = sc.textFile("hdfs:///some/path.txt")
...
rdd2: org.apache.spark.rdd.RDD[String] = ...
```

如果 Spark 在本地模式下运行，可以用 `textFile` 方法访问本地文件系统上的路径。如果输入是目录而不是单个文件，Spark 会把该目录下所有的文件作为 RDD 的输入。最后请注意，实际上 Spark 并未将数据读取到客户端机器或集群内存中。当需要对分区内的对象进行计算时，Spark 才会读入输入文件的某个部分（也称“**切片**”），然后应用其他 RDD 定义的后续转换操作（过滤和汇总等）。

我们的记录关联数据存储在一个文本文件中，文件中每行代表一个样本。我们用 `SparkContext` 的 `textFile` 方法来得到 RDD 形式的数据引用：

```
val rawblocks = sc.textFile("linkage")
...
rawblocks: org.apache.spark.rdd.RDD[String] = ...
```

这几行代码有几点值得我们注意。第一，我们声明了一个名叫 `rawblocks` 的新变量。从 shell 中可以看出，`rawblocks` 变量的类型为 `RDD[String]`，而我们并没有在变量声明时指出变量类型。这个功能在 Scala 编程语言中称为**类型推断**，它为我们写代码时节省了许多键盘输入。Scala 会尽可能从上下文中分析出变量类型。在我们的示例中，Scala 会查找 `SparkContext` 对象 `textFile` 函数的返回值类型，发现该函数返回 `RDD[String]` 类型，于是就将 `RDD[String]` 类型赋给 `rawblocks` 变量。

只要在 Scala 中定义新变量，就必须在变量名称前加上 `val` 或 `var`。名称前带 `val` 的变量是不可变变量。一旦给不可变变量赋完初值，就不能改变它，让它指向另一个值。而以 `var` 开头的变量则可以改变其指向，让它指向同一类型的不同对象。试试看如下代码的执行情况：

```
rawblocks = sc.textFile("linkage")
...
<console>: error: reassignment to val

var varblocks = sc.textFile("linkage")
varblocks = sc.textFile("linkage")
```

试图将关联数据重新赋给 `rawblocks val` 变量会报错，但重新给 `varblocks var` 变量赋值则没有问题。在 Scala REPL 中，对 `val` 变量有个例外，因为 Scala REPL 允许我们重新声明相同的不可变变量，请看代码：

```
val rawblocks = sc.textFile("linakge")
val rawblocks = sc.textFile("linkage")
```

示例中第二次声明 `rawblocks val` 变量并没有报错。这在常规 Scala 代码中是非法的，但在 shell 中却没有问题，本书的许多例子中会用到该功能。

REPL 与编译

除了交互式 shell，Spark 也支持编译程序。我们通常推荐使用 Apache Maven（https://maven.apache.org）来编译程序和管理依赖关系。本书在 GitHub 的资料库的 simplesparkproject/ 目录（https://github.com/sryza/aas/tree/master/simplesparkproject）下包含了一个完整的 Maven 工程，你可以用它作为开端。

现在你有两个选择：shell 和编译程序，但测试和构建数据处理程序时该选哪个呢？通常在初始阶段工作可能全部用 REPL 完成。REPL 可以加快原型开发，使迭代更快，很快看到你的想法的结果。但随着程序越来越大，在一个文件中维护大量代码就变得很笨拙了，这时解释 Scala 程序也要消耗更多时间。如果数据量巨大，情况会更糟，经常会出现一个操作导致 Spark 应用崩溃或 `SparkContext` 不可用的情况。如果发生这

种情况，意味着所有的工作和输入的代码都丢失了。这时我们往往应该采用混合模式。最前面的开发工作在 REPL 里完成，随着代码逐渐成熟，将代码移到编译库里。可以在 spark-shell 中引用已编译好的 JAR，只要给 spark-shell 设置 --jars 命令行参数即可。这样的话，如果使用得当，就不用频繁重新编译 JAR，同时 REPL 可以支持快速代码迭代和逐步成熟方式。

如何引用外部的 Java 和 Scala 类库呢？要编译引用了外部类库的代码，需要在工程的 Maven 配置文件（pom.xml）中指定所需的类库。要运行依赖外部类库的代码，需要在 Spark 进程中通过 classpath 将所需类库的 JAR 文件包含进来。为此，一种好的做法是使用 Maven 来打包 JAR，使生成的 JAR 包含应用程序的所有依赖文件。接着在启动 shell 时通过 --jars 属性引用该 JAR。这种方法的优点是依赖只需要在 Maven 的 pom.xml 中指定一次即可。如何进行设置，请参考本书 GitHub 资料库的 simplesparkproject/ 目录。

如果想使用第三方 Maven 仓库的某个 JAR，可以通过 --package 命令行参数告知 spark-shell 这个 JAR 的 Maven 坐标，随后 spark-shell 就会加载这个 JAR。举个例子，为加载 Scala 2.11 版本的 Wisp Visualization 库，你需要将 --packages "com.quantifind:wisp_2.11:0.0.4" 这个参数传递给 spark-shell。如果所需的 JAR 未存储在 Maven 中央仓库中，可以通过 --repositories 参数来告诉 Spark，这个 JAR 在哪个仓库中。如果想用 --packages 和 --repositories 来加载多个 JAR，可以使用逗号对多个包名或仓库名进行分隔。

2.5 把数据从集群上获取到客户端

RDD 有许多方法，我们可以用这些方法从集群读取数据到客户端机器上的 Scala REPL 中。其中最简单的方法可能就是 first 了，该方法向客户端返回 RDD 的第一个元素：

```
rawblocks.first
...
res: String = "id_1","id_2","cmp_fname_c1","cmp_fname_c2",...
```

first 方法可用于对数据集做常规检查，但通常我们更想返回更多样例数据供客户端分析。如果知道 RDD 只包含少量记录，可以用 collect 方法向客户返回一个包含所有 RDD 内容的数组。因为我们还不知道这个关联数据集有多大，所以暂时不那么做了。

还可以用 take 方法，这个方法在 first 和 collect 之间做了一些折衷，可以向客户端返回一个包含指定数量记录的数组。我们来看看如何使用 take 方法获取记录关联数据集的前 10 行记录：

```
val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...
```

```
head.length
...
res: Int = 10
```

动作

创建 RDD 的操作并不会导致集群执行分布式计算。相反，RDD 只是定义了作为计算过程中间步骤的逻辑数据集。只有调用 RDD 上的 action（动作）时分布式计算才会执行。举个例子，`count` 动作返回 RDD 中对象的个数：

```
rdd.count()
14/09/10 17:36:09 INFO SparkContext: Starting job: count ...
14/09/10 17:36:09 INFO SparkContext: Job finished: count ...
res0: Long = 4
```

`collect` 动作返回一个包含 RDD 中所有对象的 `Array`（数组）：

```
rdd.collect()
14/09/29 00:58:09 INFO SparkContext: Starting job: collect ...
14/09/29 00:58:09 INFO SparkContext: Job finished: collect ...
res2: Array[(Int, Int)] = Array((4,1), (1,1), (2,2))
```

动作不一定向本地进程返回结果。`saveAsTextFile` 动作将 RDD 的内容保存到持久化存储（比如 HDFS）上：

```
rdd.saveAsTextFile("hdfs:///user/ds/mynumbers")
14/09/29 00:38:47 INFO SparkContext: Starting job:
saveAsTextFile ...
14/09/29 00:38:49 INFO SparkContext: Job finished:
saveAsTextFile ...
```

该动作创建一个目录并为每个分区输出一个文件。切换到 Spark shell 外面的命令行，执行如下操作：

```
hadoop fs -ls /user/ds/mynumbers

-rw-r--r--   3 ds supergroup       0 2014-09-29 00:38 myfile.txt/_SUCCESS
-rw-r--r--   3 ds supergroup       4 2014-09-29 00:38 myfile.txt/part-00000
-rw-r--r--   3 ds supergroup       4 2014-09-29 00:38 myfile.txt/part-00001
```

记住，`textFile` 接受包含一个文本文件的目录作为输入，这意味将来的 Spark 作业可以把 `mynumbers` 作为其输入目录。

Scala REPL 返回的数据原始形式可能有点儿难以读懂，特别是对于包含了许多元素的数组更是如此。为了更容易读懂数组的内容，我们可以用 `foreach` 方法并结合 `println` 来打印出数组中的每个值，并且每一行打印一个值：

```
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2",
```

```
  "cmp_sex","cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"
37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,?,1,1,1,1,1,TRUE
25965,64753,1,?,1,?,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
39932,40902,1,?,1,?,1,1,1,1,1,TRUE
```

本书经常用到 `foreach(println)` 模式。它是一个常见的函数式编程模式：把函数 `println` 作为参数传递给另一个函数以执行某个动作。用过 R 的数据科学家很熟悉这种编程风格。为了在处理向量和列表时避免循环，他们习惯用高阶函数，比如 `apply` 和 `lapply`。Scala 的集合与 R 的列表和向量类似，我们通常希望少用 `for` 循环，而在处理集合元素时采用高阶函数。

很快我们就发现数据有几个问题，这些问题必须在开始对数据分析前解决好。首先，CSV 文件有一个标题行需要过滤掉，以免影响后续分析。我们可以将标题行中出现的 `"id_1"` 字符串作为过滤条件，编写一个简单的 Scala 函数来测试一行记录中是否包含该字符串，代码如下：

```
def isHeader(line: String) = line.contains("id_1")
isHeader: (line: String)Boolean
```

和 Python 类似，Scala 声明函数用关键字 `def`。和 Python 不同，我们必须为函数指定参数类型：在示例中，我们指明 `line` 参数是 `String`。函数体部分调用 `String` 类的 `contains` 方法，用于测试字符串中是否出现 `"id_1"` 字符序列，等号后的部分都是函数体的内容。虽然我们必须指定 `line` 参数的类型，但是没必要指定函数的返回类型，原因在于 Scala 编译器能根据 `String` 类的信息和 `String` 类 `contains` 方法返回 `true` 或 `false` 这一事实来推断出函数的返回类型。

有时候我们希望能显式地指明函数返回类型，特别是碰到函数体很长、代码复杂并且包含多个 `return` 语句的情况。这时候，Scala 编译器不一定能推断出函数的返回类型。为了函数代码可读性更好，也可以指明函数的返回类型。这样他人在阅读代码的时候，就不必重新把整个函数读一遍了。可以紧跟在参数列表后面声明返回类型，示例如下：

```
def isHeader(line: String): Boolean = {
  line.contains("id_1")
}
isHeader: (line: String)Boolean
```

通过用 Scala 的 `Array` 类的 `filter` 方法打印出结果，可以在 `head` 数组上测试新编写的 Scala 函数：

```
head.filter(isHeader).foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1",...
```

看起来我们的 isHeader 方法没什么问题：通过 filter 方法将 isHeader 作用在 head 数组上，返回的唯一结果是标题行本身。当然，我们其实想要的是所有非标题行。为了完成这个目标，Scala 有几种方法。第一个就是利用 Array 类的 filterNot 方法：

```
head.filterNot(isHeader).length
...
res: Int = 9
```

还可以利用 Scala 对匿名函数的支持，在 filter 函数里面对 isHeader 函数取非：

```
head.filter(x => !isHeader(x)).length
...
res: Int = 9
```

Scala 的匿名函数有点儿类似于 Python 的 lambda 函数。在示例代码中我们定义了一个名为 x 的参数并把它传给 isHeader 函数，再对 isHeader 函数的返回值取非。请注意，样例代码中没必要指定 x 变量的类型信息，Scala 编译器能够根据 head 的类型是 Array[String] 推断出 x 是 String 类。

Scala 程序员最讨厌的就是键盘输入。因此 Scala 设计了许多小功能来减少输入，比如在匿名函数的定义中，为了定义匿名函数并给参数指定名称，只输入了字符 x=>。但像这么简单的匿名函数，甚至都没必要这么做：Scala 允许使用下划线（_）表示匿名函数的参数，因此我们可以少输入 4 个字符：

```
head.filter(!isHeader(_)).length
...
res: Int = 9
```

有时这种缩写语法使代码更易阅读，因为它省略了明显多余的标识符，但有时也会使代码更难懂。代码到底是更易懂还是更难懂，这就要靠我们自己判断了。

2.6 把代码从客户端发送到集群

刚才我们见识了 Scala 语言定义和运行函数的多种方式。我们执行的代码都作用在 head 数组中的数据上，这些数据都在客户端机器上。现在，我们打算在 Spark 里把刚写好的代码应用到关联记录数据集 RDD rawblocks，该数据集在集群上的记录有数百万条。

下面是一段示例代码，是不是觉得特别熟悉？

```
val noheader = rawblocks.filter(x => !isHeader(x))
```

用于过滤集群上整个数据集的语法和过滤本地机器上的 `head` 数组的语法一模一样。可以用 `noheader` 这个 RDD 来验证过滤规则是否正确：

```
noheader.first
...
res: String = 37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
```

这太强大了！它意味着我们可以先从集群采样得到小数据集，在小数据集上开发和调试数据处理代码，等一切就绪后再把代码发送到集群上处理完整的数据集就可以了。最厉害的是，我们从头到尾都不用离开 shell 界面。除了 Spark，还真没有哪种工具能给你这种体验。

在后面几节中，我们将运用这种本地开发加测试和集群运算的方式，来展示更多处理和分析记录关联数据的技术。如果你现在想停下来喝口水，感叹一下 Spark 这个新世界的美妙，我们当然能理解。

2.7 从RDD到DataFrame

本书第 1 版的这一节中，我们介绍了一个新功能：如何在 REPL 中混合使用本地开发和测试以及集群中的运算。我们写了一段代码，对存储记录关联数据的 CSV 文件进行解析。代码完成的工作包括：把每一行按照逗号分隔成多个字段，再将每个字段转换成对应的数据类型（整型或双精度浮点数），并处理非法值。Spark 的这种处理数据的方式很有吸引力，特别是在数据集的结构不同寻常或者非标准的情况下，此时其他处理方式并不适用。

不过，我们遇到的大部分数据集都有着合理的结构，要么因为它们本来如此（比如来自数据库的表），要么因为有人已经对数据做好了清洗和结构化。对这类数据，我们完全没有必要花费精力自己写一套代码来解析它，只需简单地调用现成的类库，并利用数据的结构，即可将其解析成所需结构，然后就可以做数据分析了。Spark 1.3 中引入了一个这样的新数据结构——DataFrame。

DataFrame 是一个构建在 RDD 之上的 Spark 抽象，它专门为结构规整的数据集而设计，DataFrame 的一条记录就是一行，每行都由若干个列组成，每一列的数据类型都有严格定义。可以把 DataFrame 类型实例理解为 Spark 版本的关系数据库表。DataFrame 这个名字可能会让你联想到 R 语言的 `data.frame` 对象，或者 Python 的 `pandas.DataFrame` 对象，但是 Spark 的 DataFrame 与它们有很大的不同。这么说是因为 DataFrame 代表集群中的一个分布式数据集，而不是所有数据都存储在同一台机器上的本地数据。虽说 Spark 的 DataFrame 与 `data.frame` 以及 `pandas.DataFrame` 的用法比较相似，而且在生态系统中的角色也类似，但是 R 和 Python 中的某些操作在 Spark 中却无法使用。所以，最好把它们看成独立的实体，并尝试接纳这些不同点。

要为记录关联数据集建立一个 DataFrame，我们需要用到 SparkSession 对象 spark，这个对象是在启动 Spark REPL 时创建的：

```
spark
...
res: org.apache.spark.sql.SparkSession = ...
```

SparkSession 替代了 SQLContext，SQLContext 最初是在 Spark 1.3 中引入的，现在已经不用了。与 SQLContext 类似，SparkSession 是 SparkContext 对象的一个封装，你可以通过 SparkSession 直接访问到 SparkContext：

```
spark.sparkContext
...
res: org.apache.spark.SparkContext = ...
```

可以看到，spark.sparkContext 的值与我们之前用来创建 RDD 的 sc 变量是完全一样的。要创建一个 DataFrame，我们可以使用 SparkSession 的 Reader API 的 csv 方法：

```
val prev = spark.read.csv("linkage")
...
prev: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string, ...
```

默认情况下，CSV 文件中的每一列都是 string 类型，列名默认为 _c0、_c1、_c2，等等。要想知道一个 DataFrame 的前几行，可以调用 DataFrame 的 show() 方法。

```
prev.show()
```

可以看到，第一行是 DataFrame 表头的列名。正如我们所料，CSV 文件被整齐地划分成多个列。我们还发现有些列中出现了“?”，接下来需要将所有“?”标记为缺失值。除了给每一列正确命名以外，如果 Spark 还能推断出每一列的数据类型，那就再好不过了。

幸运的是，Spark 的 CSV 读取器提供了该项功能，我们只需设置 Spark 的 CSV reader API 即可。你可以在 spark-csv 项目的 GitHub 页面（https://github.com/databricks/spark-csv#features）上看到所有需设置的选项，spark-csv 在 Spark 1.*x* 时代是独立的项目，到了 Spark 2.*x* 才整合进来。现在我们可以通过以下方式读取并解析关联数据了：

```
val parsed = spark.read.
  option("header", "true").
  option("nullValue", "?").
  option("inferSchema", "true").
  csv("linkage")
```

对 parsed 调用 show 方法，可以看到列名已经设置成功，“?”也替换成了 null 值。要了解每列的推测类型，我们可以输出经过解析的 DataFrame 的模式信息，示例如下：

```
parsed.printSchema()
...
root
 |-- id_1: integer (nullable = true)
 |-- id_2: integer (nullable = true)
 |-- cmp_fname_c1: double (nullable = true)
 |-- cmp_fname_c2: double (nullable = true)
...
```

每个 StructField 实例包含了列名、每条记录中数据的最具体的类型，以及一个表示此列是否允许空值的布尔字段（默认值为真）。为了完成模式推断，Spark 需要遍历数据集**两次**：第一次找出每列的数据类型，第二次才真正进行解析。如果预先知道某个文件的模式，你可以创建一个 org.apache.spark.sql.types.StructType 实例，并使用模式函数将它传给 Reader API。在数据量很大的情况下，这样做可以获得巨大的性能提升，因为 Spark 不需要为确定每列的数据类型而额外遍历一次数据。

DataFrame 与数据格式

通过 DataFrameReader 和 DataFrameWriter API，Spark 2.0 内置支持多种格式读写 DataFrame。除了我们这里讨论过的 CSV 格式以外，还可以读写如下几种数据源。

json
: 支持 CSV 格式具有的模式推断功能。

parquet和orc
: 两种二进制列式存储格式，这两种格式可以相互替代。

jdbc
: 通过 JDBC 数据连接标准连接到关系型数据库。

libsvm
: 一种常用于表示特征稀疏并且带有标号信息的数据集的文本格式。

text
: 文件的每行作为字符串整体映射到 DataFrame 的一列。

要访问 DataFrameReader API 中的方法，可以调用 SparkSession 实例的 read 方法。要从文件中加载数据，可以调用 format 和 load 方法，也可以使用更快捷的方法，这些方法格式是内置的，示例如下：

```
val d1 = spark.read.format("json").load("file.json")
val d2 = spark.read.json("file.json")
```

在这个例子中，d1 和 d2 引用的底层数据是同一份 JSON 数据，因此 d1 和 d2 内容相同。每个不同的文件格式都有它们自己的设置项，可以通过 option 方法设置这些选项，参见我们对 CSV 文件的方法设置。

如果要把数据导出，你可以通过调用任何 DataFrame 实例的 write 方法访问 DataFrameWriter API。DataFrameWriter API 支持与 DataFrameReader API 相同的内置格式，所以要把文件保存成 parquet 格式的话，以下两种方法都可以：

```
d1.write.format("parquet").save("file.parquet")
d1.write.parquet("file.parquet")
```

默认情况下，Spark 在保存 DataFrame 时，如果目标文件已存在，Spark 会抛出一个错误信息。你可以通过 DataFrameWriter API 的枚举类型 SaveMode，控制 Spark 在这种情况下的行为。你可以选择强制覆盖（Overwrite）、在文件末尾追加（Append），或者文件已存在时跳过这次写入（Ignore）：

```
d2.write.mode(SaveMode.Ignore).parquet("file.parquet")
```

你也可以用一个字符串（"overwrite"、"append"、"ignore"）来指定 SaveMode，就像用 R 和 Python 的 DataFrame API 时一样。

2.8 用DataFrame API来分析数据

Spark 的 RDD API 为分析数据提供了少量易用的方法，例如 count() 方法可以计算一个 RDD 包含的记录数，countByValue() 方法可以获取不同值的分布直方图，RDD[Double] 的 stats() 方法可以获取一些概要统计信息，例如最小值、最大值、平均值和标准差。但是 DataFrame API 的工具比 RDD API 更强大。用惯了 R、Python 和 SQL 的数据科学家对这套工具应该不会觉得陌生。本节将开始探索这套接口，并将它们应用在记录关联数据上。

研究一下 DataFrame 对象实例 parsed 的模式，看一下前几行数据，我们可以看到以下特征。

- 前两个字段是整型 ID，代表在记录中匹配到的患者。
- 后面 9 个值是数值类型（双精度浮点数或整型，可能有缺失值），代表患者记录数据中不同字段的匹配得分值，如名字、生日和住址。这些字段如果只有匹配和不匹配两种情况，则用 1 表示匹配，0 表示未匹配；如果有部分匹配的情况，则用双精度浮点数表示。
- 最后一个字段是布尔值（true 或 false），表示这条记录中的一对患者是否匹配。我们的目标是创建一个简单的分类器，它可以根据患者数据中的匹配评分来预测一条记录是否匹配。我们先调用 count 方法来了解记录数，该方法在 DataFrame 和 RDD 中是完全相同的：

```
parsed.count()
...
res: Long = 5749132
```

这个数据集相对较小，小到能存放在集群中一个节点的内存上。在没有集群可用的情况

下，甚至可以存放在本地机器的内存里。到目前为止，我们每次处理数据集中的数据时，Spark 得重新打开文件，再重新解析每一行，然后才能执行所需的操作，例如显示前几行或计算记录的总数。当我们需要执行另外一个操作时，Spark 会反复执行读取及解析操作，即使我们已经从数据集中过滤出少量的数据，或者对原始数据集已经做过聚合。

这种方式浪费了计算资源。数据一旦被解析完，我们就可以把解析后的数据保存在集群中，这样就不必每次都重新解析数据了。Spark 支持这种用例，它允许我们调用 `cache` 方法，告诉 RDD 或 DataFrame 在创建时将它缓存在内存中。现在尝试缓存 `parsed`：

```
parsed.cache()
```

缓存

虽然默认情况下 DataFrame 和 RDD 的内容是临时的，但是 Spark 提供了一种持久化底层数据的机制：

```
cached.cache()
cached.count()
cached.take(10)
```

在上述代码中，调用 `cache` 方法指示在下次计算 DataFrame 时，要把 DataFrame 的内容缓存起来。在这个示例中，DataFrame 的内容是调用 `count` 方法时得到的，`take` 方法返回 DataFrame 的一个本地 `Array[Row]`，它表示前 10 个元素。当调用 `take` 时，访问的是缓存，而不是从 `cached` 的依赖关系中重新计算出来的。

Spark 为持久化数据定义了几种不同的机制，用不同的 `StorageLevel` 值表示。`cache()` 是 `persist(StorageLevel.Memory)` 的简写，它将所有 `Row` 对象存储为未序列化的 Java 对象。当 Spark 预计内存不够存放一个分区时，它干脆就不在内存中存储这个分区，这样在下次需要时就必须重新计算。在对象需要频繁访问或低延访问时，适合使用 `StorageLevel.MEMORY`，因为它可以避免序列化的开销。相比其他选项，`StorageLevel.MEMORY` 的问题是要占用更大的内存空间。另外，大量小对象会对 Java 的垃圾回收施加压力，会导致程序停顿和常见的速度缓慢问题。

Spark 也提供了 `MEMORY_SER` 的存储级别，用于在内存中分配大字节缓冲区，以存储记录的序列化内容。如果使用得当（稍后会详细介绍），序列化数据占用的空间往往约为未经序列化数据的 17%~33%。

Spark 也可以用磁盘来缓存数据。存储级别 `MEMORY_AND_DISK` 和 `MEMORY_AND_DISK_SER` 分别类似于 `MEMORY` 和 `MEMORY_SER`。对于 `MEMORY` 和 `MEMORY_SER`，如果一个分区在内存里放不下，整个分区都不会放入内存。对于 `MEMORY_AND_DISK` 和 `MEMORY_AND_DISK_SER`，如果分区在内存里放不下，Spark 会将其溢写到磁盘上。

虽然 DataFrame 和 RDD 都可以被缓存，但是有了 DataFrame 的模式信息，Spark 就可以利用数据的详细信息，帮助 DataFrame 在持久化数据时达到比使用 RDD 的 Java 对象高得多的效率。

> 决定何时缓存数据是一门艺术，这个决定通常涉及空间和速度之间的权衡，而且还要时不时受到垃圾收集器的影响，因此如何抉择是很复杂的事情。一般来说，当数据可能被多个操作依赖时，并且相对于集群可用的内存和磁盘空间而言，如果数据集较小，而且重新生成的代价很高，那么数据就应该被缓存起来。

数据缓存完后，接下来我们想要知道，记录中匹配记录相对于不匹配记录的比例。在使用 RDD API 的情况下，我们需要编写一个 Scala 内联函数，从每个记录中提取列 `is_match` 的值，得到一个 `RDD[Boolean]`，然后调用 `countByValue` 函数来统计 `true` 和 `false` 出现的频率，计算完成后将一个 `Map[Boolean, Long]` 返回给客户端。事实上，我们仍然可以对位于解析后的 DataFrame 底层的 RDD 执行这种计算。

```
parsed.rdd.
  map(_.getAs[Boolean]("is_match")).
  countByValue()
...
Map(true -> 20931, false -> 5728201)
```

DataFrame 封装的 RDD 由 `org.apache.spark.sql.Row` 的实例组成，包括通过索引位置（从 0 开始计数）获取每个记录中值的访问方法，以及允许通过名称查找给定类型的字段的 `getAs[T]` 方法。

虽然基于 RDD 的分析能获得我们想要的结果，但作为 Spark 上通用的数据分析方法，它还有许多待改进之处。首先，当数据集中仅有几个不同的值时，使用 `countByValue` 函数来进行统计是唯一的正确做法。如果有许多不同的值，那么使用一个不返回结果到客户端的 RDD 函数将更为高效，例如 `reduceByKey`。其次，如果需要在随后的计算中使用 `countByValue` 聚合函数返回的结果，那么我们需要使用 `SparkContext` 的 `parallelize` 方法将数据从客户端发回集群。一般来说，对结构化数据的聚合，我们希望有一种简单的方法可以适用于任何大小的数据集，这正是 DataFrame API 所提供的功能：

```
parsed.
  groupBy("is_match").
  count().
  orderBy($"count".desc)
  show()
...
+--------+-------+
|is_match|  count|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

我们不需要再写一个函数来解析 `is_match` 列，只需要将列名 `is_match` 传递给 DataFrame 的 `groupBy` 方法，然后调用 `count()` 方法计算每个分组的记录数，再将结果按 `count` 列降

序排序，最后简单地通过 show 方法就可以将计算结果呈现在 REPL 中。幕后的 Spark 引擎决定了如何最高效地执行聚合并返回结果，而用户无须担心底层 RDD API 使用的细节。在基于 Spark 的数据分析中，这显然是一种更简单、更快速、更有表现力的方法。

值得注意的是，我们有两种方式引用 DataFrame 的列名：作为字面量引用，例如 groupBy("is_match")；或者作为 Column 对象应用，例如 count 列上使用的特殊语法 $"<col>"。这两种方法在大多数情况下都是合法的，但是在 count 列上调用 desc 方法时需要使用 $ 语法。如果漏掉了字符串前面的 $ 符号，Scala 就会抛出一个错误，因为类 String 没有一个名为 desc 的方法。

DataFrame 的聚合函数

除了 count 方法以外，结合 DataFrame API 的 agg 方法和 org.apache.spark.sql.functions 包中定义的聚合函数，我们可以计算更复杂的聚合分析，比如总和、最小值、最大值和标准差。例如，为了求 parsed 的 cmp_sex 字段的整体均值和标准差，我们可以这样写代码：

```
parsed.agg(avg($"cmp_sex"), stddev($"cmp_sex")).show()
+-----------------+--------------------+
|     avg(cmp_sex)|stddev_samp(cmp_sex)|
+-----------------+--------------------+
|0.955001381078048|  0.2073011111689795|
+-----------------+--------------------+
```

注意，默认情况下 Spark 只计算样本标准差；要计算总体标准差，需要使用 stddev_pop 函数。

你可能已经注意到，DataFrame API 的函数很像 SQL 查询组件，这并不是一个巧合。事实上，我们可以把创建的任何 DataFrame 都看作数据库中的一张表，并且可以使用熟悉而又强大的 SQL 语法来表达我们的问题。首先，将 DataFrame 对象 parsed 所关联的表名告诉 Spark SQL 引擎，因为 parsed 这个变量名对于 Spark SQL 引擎是不可用的：

```
parsed.createOrReplaceTempView("linkage")
```

因为 parsed 这个 DataFrame 变量只在这个 Spark REPL 的会话中可用，所以 linkage 现在是一张**临时表**。Spark SQL 也可以用于查询 HDFS 中的持久性表，只要我们设置 Spark 连接 Apache Hive metastore，metastore 记录了结构化数据集的模式和位置。当临时表在 Spark SQL 引擎中注册后，我们可以这样查询它：

```
spark.sql("""
  SELECT is_match, COUNT(*) cnt
  FROM linkage
  GROUP BY is_match
  ORDER BY cnt DESC
""").show()
```

```
...
+--------+-------+
|is_match|    cnt|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

和 Python 一样，Scala 中 3 个连续的双引号可以用来表示一个跨多行的字符串。Spark 1.*x* 的 Spark SQL 编译器主要是为了兼容 HiveQL 中的非标准语法，这样用户就能比较容易地从 Apache Hive 迁移到 Spark 上来。Spark 2.0 默认使用兼容 ANSI 2003 的 Spark SQL，当然我们也可以选择使用 HiveQL 模式，只需通过 Spark Sesssion 的 Builder API 创建一个 `SparkSession` 实例，然后调用 `enableHiveSupport` 方法即可。

在 Spark 中进行数据分析，到底是应该使用 Spark SQL 还是 DataFrame API 呢？这两种方法各有利弊。SQL 大家都很熟悉，简单的查询很容易表达。在常用的列式存储中，如 ORC 和 Parquet，SQL 是快速读取和过滤存储最好的方式。SQL 的缺点是很难用动态、可读和可测试的方式来表达复杂的多阶段分析，而这些都是 DataFrame API 的强项。在本书的其余章节，Spark SQL 和 DataFrame API 二者都会使用。读者可以思考一下我们为什么这样选择，并练习如何在二者之间进行转换。

Spark SQL 与 Hive 的连接

Spark 1.*x* 有一个 `HiveContext` 类，它是 `SQLContext` 的子类，并支持 Hive 特有的 SQL 方言—— HiveQL。如果复制一个 hive-site.xml 文件到 Spark 安装目录的 conf 文件夹下，`HiveContext` 就可以跟 Hive metastore 交互了。在 Spark 2.*x* 中，虽然 `HiveContext` 被弃用了，但是仍然可以通过 hive-site.xml 文件连接到 Hive metastore，还可以调用 SparkSession Builder API 的 `enableHiveSupport` 方法来使用 HiveQL 语法进行查询。

```
val sparkSession = SparkSession.builder.
 master("local[4]")
 .enableHiveSupport()
 .getOrCreate()
```

在 Spark 2.*x* 中，你可以将 Hive metastore 中任何一张表视为一个 DataFrame，并可以使用 Spark SQL 对 metastore 中的表进行查询，还可以将这些查询结果保存在 metastore 中，这样它们就可以被其他 SQL 工具访问到了，包括 Hive 自身、Apache Impala 和 Presto 等。

2.9 DataFrame的统计信息

虽然在许多数据分析工作中，无论使用 SQL 还是 DataFrame，结果都是一样的，但是仍有一些常见工作，用 DataFrame 表示起来很简洁，而用 SQL 却显得很冗余。例如，计算一个

DataFrame 中数值列所有非空值的最小值、最大值、平均值和标准差。在 R 中，这个函数叫 summary；在 Spark 中，这个函数叫 describe，与 Pandas 中相同：

```
val summary = parsed.describe()
...
summary.show()
```

DataFrame 类型的 parsed 实例中的每个变量，在 DataFrame 类型的 summary 实例中都有相对应的一列；还有一个名为 summary 的列，用于指示 count、mean、stddev、min 和 max 这 5 个指标在行中是否出现。为了让 summary 的统计信息更便于阅读和比较，我们可以使用 select 方法来选出一部分列：

```
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
+-------+------------------+------------------+
|summary|      cmp_fname_c1|      cmp_fname_c2|
+-------+------------------+------------------+
|  count|           5748125|            103698|
|   mean|0.7129024704436274|0.9000176718903216|
| stddev|0.3887583596162788|0.2713176105782331|
|    min|               0.0|               0.0|
|    max|               1.0|               1.0|
+-------+------------------+------------------+
```

请注意，cmp_fname_c1 和 cmp_fname_c2 的 count 变量值并不相同：几乎每条记录的 cmp_fname_c1 都是非空的，而只有 2% 的记录中 cmp_fname_c2 字段是非空的。为了得到一个有用的分类器，模型依赖的那些变量在数据中不能有太多缺失值，除非这些值的缺失也表示某种与记录是否匹配相关的含义。

在对数据中变量的分布有了大体的了解后，我们就想知道这些变量与列 is_match 的值之间的相关性。因此，接下来我们对 parsed 中 is_match 字段为 true 和 false 的两个子集计算概要统计信息。我们既可以使用 SQL 风格的 where 语法，也可以使用 DataFrame API 的 Column 对象来过滤 DataFrame，然后对得到的 DataFrame 使用 describe 方法：

```
val matches = parsed.where("is_match = true")
val matchSummary = matches.describe()

val misses = parsed.filter($"is_match" === false)
val missSummary = misses.describe()
```

where 函数的字符串的内部逻辑如果放到 Spark SQL 的 WHERE 子句中，语法也是正确的。使用 DataFrame API 方式的过滤条件稍微复杂一点：我们需要对列 $"is_match" 使用 === 操作符，并且还需要用 lit 方法封装布尔文字 false，这样就可以将其转换成能与 is_match 做对比的 Column 对象。需要注意的是，where 函数是 filter 函数的一个别名，我们可以随意调换上述代码片段中的 where 和 filter，而结果不会发生任何变化。

现在我们比较 matchSummary 和 missSummary 这两个 DataFrame，这样就能知道记录的匹配

情况对变量的分布有何影响。尽管数据集相对较小，单做这种比较也没有多大意思，但其实我们真正想做的是对 matchSummary 和 missSummary 这两个 DataFrame 做一个转置，将它们的行与列调换，这样就可以将两个转置过的 DataFrame 按变量关联起来，以便分析这些概要统计信息，这种做法被大多数数据科学家称为“数据集转置”（pivoting）或“重塑”（reshaping）。下一节将展示如何在 Spark 中执行这些转换。

2.10 DataFrame的转置和重塑

为了转置概要统计信息，首先要做的是将 matchSummary 和 missSummary 这两个 DataFrame 类型实例从“宽表”转换成“长表”。宽表中行代表指标，列代表变量；长表的每一行代表一个指标、一个变量，以及指标和变量对应的值。转换完成后，我们就可以将长表形式的 DataFrame 转换成另外一个宽表形式的 DataFrame，这样就完成了转置操作，只不过这一次操作中，变量对应行，指标对应列。

将宽表转换成长表，可以利用 DataFrame 的 flatMap 方法，它是 RDD.flatMap 的一个封装。flatMap 是 Spark 中最有用的转换函数之一：它接受一个函数作为参数，该函数处理一条输入记录，并返回一个包含零条或多条输出记录的序列。你可以将 flatMap 看作我们使用过的 map 和 filter 转换函数的一般形式：map 是 flatMap 的一种特殊形式，即一条输入记录仅产生一条输出记录；filter 是 flatMap 的另一种特殊形式，即输入和输出类型相同，并且基于一个布尔函数决定返回零条或一条记录。

为了让 flatMap 方法在一般的 DataFrame 上也能工作，要用到 DataFrame 的 schema 对象来获取 DataFrame 每一列的名字：

```
summary.printSchema()
...
root
 |-- summary: string (nullable = true)
 |-- id_1: string (nullable = true)
 |-- id_2: string (nullable = true)
 |-- cmp_fname_c1: string (nullable = true)
...
```

在 summary 实例的 schema 变量中，每个字段都被视为一个字符串。要想分析数值形式的统计信息，需要将字符串转换为双精度浮点数，输出的 DataFrame 应该有 3 列：指标名称（count、mean 等）、列名（id1、cmp_by 等），以及该列统计信息的双精度值。

```
val schema = summary.schema
val longForm = summary.flatMap(row => {
  val metric = row.getString(0)
  (1 until row.size).map(i => {
    (metric, schema(i).name, row.getString(i).toDouble)
  })
})
```

以上代码片段做了很多事情，让我们逐行进行分析。对 DataFrame 类型的 summary 的每一行，我们通过调用 row.getString(0) 来依据位置获得这个指标的名称。对于这一行中位置 1 后的其他列，我们调用 flatMap 操作，生成了一个元组序列。元组中第一个条目是指标的名称，第二个条目是列的名字（通过 schema(i).name 对象获取），第三个条目是统计量的值，我们用强制类型转换将 row.getString(i) 方法获取的字符串解析成一个双精度浮点数。

toDouble 方法是隐式转换的一个实例，而**隐式类型**是 Scala 最强大（也可能最危险）的特性之一。在 Scala 中，类 String 的实例其实就是 java.lang.String，而 java.lang.String 类并没有名为 toDouble 的方法；相反，这个方法定义在一个名为 StringOps 的 Scala 类中。隐式转换的工作原理如下：当在 Scala 的对象上调用一个方法，并且 Scala 编译器没有在该对象上的类定义中找到这个方法，那么编译器就会尝试将你的对象转换成拥有这个方法的类的实例。在这种情况下，编译器会发现 Java 的 String 类没有定义 toDouble 方法，而类 StringOps 中却有这个方法，并且 StringOps 类还有一个方法可以将 String 类的实例转换成 StringOps 类的实例。编译器悄悄地将 String 对象转换成 StringOps 对象，并调用新对象的 toDouble 方法。

Scala 类库开发人员（包括 Spark 核心开发者）非常喜欢隐式转换类型。它允许开发者增强核心类的功能，这样即使是像 String 这种不允许修改的类也可以进行增强。但对这些工具的用户来说，隐式类型转换就不那么简单了，因为隐式类型转换使得用户难以找到定义类方法的确切位置。尽管如此，我们还会在示例中遇到一些隐式转换，所以我们有必要提前熟悉一下。

这个代码块中最后需要注意的一点是变量 longForm 的类型：

```
longForm: org.apache.spark.sql.Dataset[(String, String, Double)]
```

这是我们第一次直接使用 Dataset[T] 接口，尽管我们一直在使用它的一个特例 DataFrame，DataFrame 其实是 Dataset[Row] 类型的别名。Dataset[T] 是 Spark 2.0 中新添加的 API，它是 Spark 1.3 中引入的 DataFrame 类型的一般化，能够处理比 Row 更丰富的数据类型。本章稍后将更详细地介绍 Dataset 的接口，但是现在你需要知道的是，由于 Spark API 中的一些巧妙的隐式转换，我们总是可以将 Dataset 转换回 DataFrame：

```
val longDF = longForm.toDF("metric", "field", "value")
longDF.show()
+------+------------+-------------------+
|metric|       field|              value|
+------+------------+-------------------+
| count|        id_1|          5749132.0|
| count|        id_2|          5749132.0|
| count|cmp_fname_c1|          5748125.0|
...
| count|      cmp_by|          5748337.0|
| count|     cmp_plz|          5736289.0|
```

```
|  mean|        id_1|  33324.48559643438|
|  mean|        id_2|  66587.43558331935|
|  mean|cmp_fname_c1| 0.7129024704436274|
...
|  mean|      cmp_bd|0.22446526708507172|
|  mean|      cmp_bm|0.48885529849763504|
+------+------------+-------------------+
```

给定一个 DataFrame 长表，可以得到这样一个宽表：对用作转置表行的列执行 `groupBy` 操作，然后对用作转置表列的列执行 `pivot` 操作。`pivot` 操作需要知道转置列的所有不同值，对列 `values` 使用 `agg(first)` 操作，我们就可以指定宽表中每个单元格的值，因为每个 `field` 和 `metric` 的组合都只有一个值，所以这样做是没问题的：

```
val wideDF = longDF.
  groupBy("field").
  pivot("metric", Seq("count", "mean", "stddev", "min", "max")).
  agg(first("value"))
wideDF.select("field", "count", "mean").show()
...
+------------+---------+-------------------+
|       field|    count|               mean|
+------------+---------+-------------------+
|     cmp_plz|5736289.0|0.00552866147434343|
|cmp_lname_c1|5749132.0| 0.3156278193084133|
|cmp_lname_c2|   2464.0|0.31841283153174377|
|     cmp_sex|5749132.0|  0.955001381078048|
|      cmp_bm|5748337.0|0.48885529849763504|
...
|      cmp_bd|5748337.0|0.22446526708507172|
|      cmp_by|5748337.0| 0.2227485966810923|
+------------+---------+-------------------+
```

现在我们已经知道了如何转置一个 DataFrame 类型的 `summary`，让我们将这段逻辑用一个函数实现，这样就可以在 `matchSummary` 和 `missSummary` 这两个 DataFrame 中重用了。在另一个 shell 窗口中使用文本编辑器，复制并粘贴以下代码，并保存到一个名为 Pivot.scala 的文件中：

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.first

def pivotSummary(desc: DataFrame): DataFrame = {
  val schema = desc.schema
  import desc.sparkSession.implicits._

  val lf = desc.flatMap(row => {
    val metric = row.getString(0)
    (1 until row.size).map(i => {
      (metric, schema(i).name, row.getString(i).toDouble)
    })
  }).toDF("metric", "field", "value")
```

```
    lf.groupBy("field").
      pivot("metric", Seq("count", "mean", "stddev", "min", "max")).
      agg(first("value"))
  }
```

现在在 Spark shell 中键入 `load Pivot.scala`，Scala REPL 将会动态编译你的代码，使 `pivotSummary` 函数对 `matchSummary` 和 `missSummary` 都可用：

```
val matchSummaryT = pivotSummary(matchSummary)
val missSummaryT = pivotSummary(missSummary)
```

2.11 DataFrame的连接和特征选择

到目前为止，我们只用了 Spark SQL 和 DataFrame API 来过滤和聚合一个数据集中的记录，但是也可以使用这些工具来完成 DataFrame 之间的连接（内连接、左外连接、右外连接和全连接）。尽管 DataFrame API 有一个 `join` 函数，但是用 Spark SQL 来表示这些连接会更容易，特别是当待连接的表中有许多列名在两个表中都存在时，通过 `select` 表达式能更方便地指出所引用的列。让我们为 `matchSummary` 和 `missSummary` 这两个 DataFrame 创建临时视图，在 `field` 列上连接它们，并在结果行上计算一些简单的统计信息：

```
matchSummaryT.createOrReplaceTempView("match_desc")
missSummaryT.createOrReplaceTempView("miss_desc")
spark.sql("""
  SELECT a.field, a.count + b.count total, a.mean - b.mean delta
  FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
  WHERE a.field NOT IN ("id_1", "id_2")
  ORDER BY delta DESC, total DESC
""").show()
...
+------------+---------+--------------------+
|       field|    total|               delta|
+------------+---------+--------------------+
|     cmp_plz|5736289.0|  0.9563812499852176|
|cmp_lname_c2|   2464.0|  0.8064147192926264|
|      cmp_by|5748337.0|  0.7762059675300512|
|      cmp_bd|5748337.0|   0.775442311783404|
|cmp_lname_c1|5749132.0|  0.6838772482590526|
|      cmp_bm|5748337.0|  0.5109496938298685|
|cmp_fname_c1|5748125.0|  0.2854529057460786|
|cmp_fname_c2| 103698.0| 0.09104268062280008|
|     cmp_sex|5749132.0|0.032408185250332844|
+------------+---------+--------------------+
```

好的特征有两个特点：第一，对于匹配的记录和不匹配的记录，该特征的值有明显不同（因此，平均值之间的差距是很大的）；第二，对于数据集的任何记录对，该特征通常都存在值，所以我们可以依赖它。按照这个标准，`cmp_fname_c2` 并不是很有用，因为它在很多时候都是缺失的，而且对于匹配的记录和不匹配的记录，该特征的平均值差异相对较

小——取值范围都是 0~1，而平均值之差只有 0.09。cmp_sex 特征也不是特别有用，虽然它在每对记录中都出现了，但是平均值仅相差 0.03。

相比之下，cmp_plz 和 cmp_by 这两个特征就非常好，它们几乎出现在每一对记录中，而且平均值的差值也很大（这两个特征都超过了 0.77）。cmp_bd、cmp_lname_c1 和 cmp_bm 这些特征看起来也有用：它们在数据集中总体上都是有值的，并且匹配记录的平均值和不匹配记录的平均值差值较大。

特征 cmp_fname_c1 和 cmp_lname_c2 情况比较复杂：cmp_fname_c1 区分得不是很好（均值的差值只有 0.28），但是在每对记录中几乎都不会缺席；cmp_lname_c2 的平均值相差很大，但是在很多记录中都是缺失的。基于这份数据，我们不是很清楚在什么情况下应该在我们的模型中加入这两个特征。

现在，我们将基于 cmp_plz、cmp_by、cmp_bd、cmp_lname_c1 以及 cmp_bm 这些明显很好的特征值之和，构建一个简单的评分模型，对记录的相似性进行排序。对于少数缺少这些特征值的记录，我们将在求和时使用 0 来替代 null。通过创建一个由计算出的评分与 is_match 列组成的 DataFrame，对区分匹配记录和不匹配记录的评分设置多种不同的阈值并评估效果，我们就能对这个简单模型的性能有一个大体认识了。

2.12 为生产环境准备模型

尽管我们可以把评分函数写成一个 Spark SQL 的查询，但是在很多情况下，我们希望能将评分规则或机器学习模型部署到生产环境中，在那里并没有足够的时间运行 Spark SQL 来得到答案。对于这些情况，我们希望编写和测试的函数能够在 Spark 上运行，但是生产代码并不依赖于 Spark JAR 包，也不需要运行 SparkSession 来执行代码。

为了剥离出模型中 Spark 特定的组件，我们希望有一种创建简单记录类型的方法，从而可以将 DataFrame 中的字段视作静态类型的变量，而不用在 Row 中动态查找。幸运的是，Scala 提供了一种便捷的语法来创建这些记录，称为 case 类。case 类是一个简单的不可变类，它默认实现了所有 Java 类的基本方法，例如 toString、equals 和 hashCode，使其非常容易使用。让我们为记录关联数据创建一个 case 类，其中字段名字及其类型与 DataFrame parsed 中列的名字和类型一一对应：

```
case class MatchData(
  id_1: Int,
  id_2: Int,
  cmp_fname_c1: Option[Double],
  cmp_fname_c2: Option[Double],
  cmp_lname_c1: Option[Double],
  cmp_lname_c2: Option[Double],
  cmp_sex: Option[Int],
  cmp_bd: Option[Int],
```

```
  cmp_bm: Option[Int],
  cmp_by: Option[Int],
  cmp_plz: Option[Int],
  is_match: Boolean
)
```

值得注意的是，我们使用了 Scala 内建的 Option[T] 类型来表示输入数据中字段的值是否为 null。在使用之前，Option 类需要客户端节点检查特定的字段是否为空（使用 None 对象表示），以避免 Scala 代码中抛出 NullPointerExceptions。像 id_1、id_2 和 is_match 这种不含 null 值的字段，可以不使用 Option 封装。

定义了类之后，我们就可以使用 as[T] 方法将 parsed 转换为 Dataset[MatchData]：

```
val matchData = parsed.as[MatchData]
matchData.show()
```

如你所见，matchData 这个 Dataset 中所有的列和值与 parsed 这个 DataFrame 中的数据是一样的，我们仍然可以对 matchData 使用所有 SQL 风格的 DataFrame API 方法以及 Spark SQL 代码。两者之间的主要区别是，当我们对 matchData 调用函数时，例如 map、flatMap 和 filter，我们处理的是 MatchData 这个 case 类，而不是 Row 类。

对于评分函数，我们将计算一个 Option[Double] 类型的字段（cmp_lname_c1）和 4 个 Option[Int] 类型的字段（cmp_plz、cmp_by、 cmp_bd 以及 cmp_bm）的总和。让我们写一个小助手 case 类来减少检查 Option 值是否存在的一些相关样板代码：

```
case class Score(value: Double) {
  def +(oi: Option[Int]) = {
    Score(value + oi.getOrElse(0))
  }
 }
```

case 类 Score 以一个 Double 类型的值（当前和）开始，并定义了一个 \+ 方法。该方法取 Option 中当前和的值，如果有值的话就取该选项的值，否则返回 0，这样它就将 Option[Int] 值合并到当前和中。为了使评分函数的名称更容易理解，这里我们用到了 Scala 的一个特点：对函数名称的限制没有 Java 那么死板，Scala 函数名的选择更广。

```
def scoreMatchData(md: MatchData): Double = {
  (Score(md.cmp_lname_c1.getOrElse(0.0)) + md.cmp_plz +
      md.cmp_by + md.cmp_bd + md.cmp_bm).value
}
```

实现了评分函数后，我们现在可以计算 matchData Dataset 中的每个 MatchData 对象的分数和 is_match 字段的值，并将结果存储在一个 DataFrame 中：

```
val scored = matchData.map { md =>
(scoreMatchData(md), md.is_match)
}.toDF("score", "is_match")
```

2.13 评估模型

创建评分函数的最后一步是决定分数的阈值，超过该阈值就预测这两个记录是匹配的。如果阈值设置过高，匹配的记录将被错误地标记为不匹配，这种情况称为**假阴性率**（false-negative rate）；而如果阈值设置过低，不匹配的记录将被错误地标记为匹配，这种情况称为**假阳性率**（false-positive rate）。对于任何非平凡的问题，我们总是需要在假阳性和假阴性之间进行取舍，而阈值的设置通常与模型应用的实际情况有关，需要在两种错误的相对代价之间进行权衡。

为了帮助我们选择阈值，可以创建一个 2×2 的**关联表** [也称为**交叉制表**（cross tabulation）或**交叉表**（crosstab）]，它统计记录中分数低于 / 高于阈值的记录数，以及两个类别中匹配 / 不匹配的记录数。因为尚不知道将要使用的阈值，所以我们要编写一个函数，它使用 DataFrame API，并以 `scored` 这个 DataFrame 和选择的阈值作为参数计算交叉表：

```
def crossTabs(scored: DataFrame, t: Double): DataFrame = {
  scored.
    selectExpr(s"score >= $t as above", "is_match").
    groupBy("above").
    pivot("is_match", Seq("true", "false")).
    count()
}
```

注意，我们引入了 DataFrame API 的 `selectExpr` 方法，基于 `t` 参数的值动态地决定字段 `above` 的值，这里使用了 Scala 的字符串替换语法。Scala 字符串替换语法允许我们使用名称替换变量，只要在字符串字面量前加上一个字母 `s`（又一个 Scala 隐式技巧带来的便利）。定义完上述字段，我们就可以创建一个通常由 `groupBy`、`pivot` 和 `count` 方法三者组合而成的交叉表。

使用一个较高的阈值 4.0，意味着 5 个特征的平均值为 0.8，可以过滤掉几乎所有的不匹配记录，同时保留 90% 的匹配记录。

```
crossTabs(scored, 4.0).show()
...
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20871|    637|
|false|   60|5727564|
+-----+-----+-------+
```

使用一个较低的阈值 2.0，可以保证捕捉到**所有**已知的匹配记录，但代价是假阳性（见右上方的单元格）很高：

```
crossTabs(scored, 2.0).show()
...
```

```
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20931| 596414|
|false| null|5131787|
+-----+-----+-------+
```

尽管假阳性的数量有点儿多，但这个比较宽松的过滤器仍然去掉了 90% 的不匹配记录，同时保留了所有的真正匹配。虽然这已经很好了，但是还有改进空间；读者可以试一下能否找到一个更好的评分函数，这个函数要能成功地识别每一个真正的匹配，并且保持假阳性少于 100 个，它可以使用来自 `MatchData` 的其他值（包括缺失和不缺失的）。

2.14 小结

在阅读本章之前，你可能还没有用 Scala 和 Spark 做过数据准备和分析；或者你已经熟悉 Spark 1.0 API，现在正在努力学习 Spark 2.0 中的新技术。无论何种情况，我们都希望你能体会到这些工具提供的强大支持。如果你已经有了使用 Scala 和 Spark 的经验，我们希望你把本章介绍给你的朋友和同事，让他们也了解 Scala 和 Spark 的强大之处。

本章的目标是为你提供足够的 Scala 知识，以便能够理解并完成本书中的其他实例。如果你习惯通过实例来学习，那你得继续看看后面几章，届时将介绍 Spark 的机器学习库 MLlib。

当你成为资深 Spark 和 Scala 数据分析人员时，可能需要开始构建工具和类库，以帮助其他分析师和数据科学家应用 Spark 来解决问题。此时看看 Scala 其他的书会对你的开发有所帮助，比如由 Deam Wampler 和 Alex Payne 所著的《Scala 程序设计》[1]，以及 Alvin Alexander 所著的 *Scala CookBook*（这两本书的英文版均由 O'Reilly 出版社出版）。

注 1：该书第 2 版已经由人民邮电出版社出版，书号 9787115416810。——编者注

第3章

音乐推荐和Audioscrobbler数据集

作者：肖恩·欧文

偏好是无法度量的。

——佚名

经常有人问起我的职业。“数据科学”或“机器学习”固然听起来很高端，但常常把对方搞得一头雾水。发生这种情况很正常，即使数据科学家自己也很难把数据科学说清楚。数据科学就是存储大量数据，对数据进行计算，然后进行预测吗？通常这时我会直接举个例子来帮助提问者搞清楚我到底是做什么的：“嗯，你在亚马逊网站买了书以后，它会向你推荐类似的书，对吗？对，就是这个意思，其实它就用到了数据科学！”

从经验上来讲，推荐引擎大体上属于大规模机器学习。大家对此都了解，而且大部分人在亚马逊上都见过。从社交网络到视频网站，再到在线零售，都用到了推荐引擎，大家也都知道推荐引擎。实际应用中的推荐引擎我们也能直接看到。虽然我们知道 Spotify 上是计算机在挑选播放的歌曲，但我们可不一定知道 Gmail 系统可以判断收件箱里的邮件是不是垃圾邮件。

相比其他的机器学习算法，推荐引擎的输出更直观，更容易理解。有时这甚至会让人很激动。尽管我们认为每个人的音乐喜好都非常个性化，并且也很难解释这种现象，但是推荐引擎却很擅长推荐一些让人喜爱的歌曲，这些歌曲连我们自己都不知道会喜欢。

最后，在推荐引擎应用比较广泛的领域，比如音乐和电影，要解释为什么推荐的音乐和一个人以前听过的音乐相吻合，这是相对比较容易的。但对某些聚类和分类算法来说，情况

就不是这样了。比如，支持向量机分类器其实就是一组系数，用这个分类器进行预测时，即使是业内人士，也很难解释这些系数的意义。

现在该开始介绍接下来的 3 章了，这 3 章讲述 Spark 中主要的机器学习算法。其中一章围绕推荐引擎展开，主要介绍音乐推荐。在随后的章节中我们先介绍 Spark 和 MLlib 的实际应用，接着介绍一些机器学习的基本思想，这样的阐述方式读者接受起来比较容易。

3.1 数据集

本章示例使用 Audioscrobbler 公开发布的一个数据集。Audioscrobbler 是 last.fm 的第一个音乐推荐系统。last.fm 创建于 2002 年，是最早的互联网流媒体广播站点之一。Audioscrobbler 提供了开放的“scrobbling”API，“scrobbling”可以记录听众播放过哪些艺术家的歌曲。last.fm（https://www.last.fm/）使用这些音乐播放记录构建了一个强大的音乐推荐引擎。由于第三方应用和网站可以把音乐播放数据反馈给这个推荐引擎，这个推荐引擎系统覆盖了数百万的用户。

在 last.fm 的年代，推荐引擎方面的研究大多局限于评分类数据。换句话说，人们常常把推荐引擎看成处理“Bob 给 Prince 的评价是 3 星半”这类输入数据的工具。

Audioscrobbler 数据集有些特别，因为它只记录了播放数据，如“Bob 播放了一首 Prince 的歌曲”。播放记录所包含的信息比评分要少。仅仅凭 Bob 播放过某歌曲这一信息并不能说明他真的喜欢这首歌。有时候我们会随便打开一首歌，甚至是整张专辑，然后就离开了房间，可能都不关心歌到底是谁唱的。

然而，人们虽然经常听音乐，但很少给音乐评分。因此 Audioscrobbler 数据集要大得多，它覆盖了更多的用户和艺术家，也包含了更多的总体信息，虽然单条记录的信息比较少。这种类型的数据通常被称为**隐式反馈数据**，因为用户和艺术家的关系是通过其他行动隐含体现出来的，而不是通过显式的评分或点赞得到的。

2005 年 last.fm 发布了该数据集的一个版本，读者可以在网上下载到压缩的归档文件（http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html）[1]。下载归档文件后，你会发现里面有几个文件。主要的数据集在文件 user_artist_data.txt 中，它包含 141 000 个用户和 160 万个艺术家，记录了约 2420 万条用户播放艺术家歌曲的信息，其中包括播放次数信息。

数据集在 artist_data.txt 文件中给出了每个艺术家的 ID 和对应的名字。请注意，记录播放信息时，客户端应用提交的是艺术家的名字。名字如果有拼写错误，或使用了非标准的名称，事后才能被发现。比如，“The Smiths”“Smiths, The”和“the smiths”看似代表不同

注 1：若此链接无法下载，请访问 http://www.iro.umontreal.ca/~lisa/datasets/profiledata_06-May-2005.tar.gz。——译者注

艺术家的 ID，但它们其实明显是指同一个艺术家。因此，为了将拼写错误的艺术家 ID 或 ID 变体对应到该艺术家的规范 ID，数据集提供了 artist_alias.txt 文件。

3.2 交替最小二乘推荐算法

现在我们要给这个隐式反馈数据选择一个合适的推荐算法。这个数据集只记录了用户和歌曲之间的交互情况。除了艺术家名字外，数据集没有包含用户的信息，也没有提供歌手的其他任何信息。我们要找的学习算法不需要用户和艺术家的属性信息。这类算法通常称为协同过滤算法（https://en.wikipedia.org/wiki/Collaborative_filtering）。举个例子，根据两个用户的年龄相同来判断他们可能有相似的偏好，这**不叫**协同过滤。相反，根据两个用户播放过许多相同歌曲来判断他们可能都喜欢某首歌，这**才叫**协同过滤。

Audioscrobbler 数据集包含了数千万条某个用户播放了某个艺术家歌曲次数的信息，看起来是很大。但从另一方面来看数据集又很小而且不充足，因为数据集是稀疏的。虽然数据集覆盖 160 万个艺术家，但平均来算，每个用户只播放了大约 171 个艺术家的歌曲。有的用户只播放过一个艺术家的歌曲。对这类用户，我们也希望算法能给出像样的推荐。毕竟每个用户在某个时刻只能播放一首歌曲。

最后，我们希望算法的扩展性好，不但能用于构建大型模型，而且推荐速度快。我们通常都要求推荐是接近实时的，也就是在一秒内给出推荐，而不是要等一天。

本实例将用到潜在因素（https://en.wikipedia.org/wiki/Factor_analysis）模型中的一种模型，这类模型涉及的范围很广泛。潜在因素模型试图通过数量相对少的**未被观察到的底层原因**，来解释大量用户和产品之间**可观察到的交互**。打个比方：有几千个专辑可选，为什么数百万人偏偏只买其中某些专辑？可以用对类别（可能只有数十种）的偏好来解释用户和专辑的关系，其中偏好信息并不能直接观察到，而数据也没有给出这些信息。

比如说，有一位客户购买了重金属乐队 Megadeth 和 Pantera 的专辑，同时还购买了古典音乐家莫扎特的专辑。很难解释为什么该客户只购买这些专辑，而没有买其他的。然而这也可能只是冰山一角，也许该客户喜欢的风格很广泛——从重金属到前卫摇滚，再到古典音乐。这种解释更简单，而且这样解释比较有利的一点是，客户可能对其他类型的专辑也感兴趣。在这个例子中，“喜欢重金属、前卫摇滚和古典音乐”这 3 个潜在因素可以解释数以万计的人对专辑的偏好。

说得更明确一些，本实例用的是一种矩阵分解模型（https://en.wikipedia.org/wiki/Non-negative_matrix_factorization）。数学上，这些算法把用户和产品数据当成一个大矩阵 A，矩阵第 i 行和第 j 列上的元素有值，代表用户 i 播放过艺术家 j 的音乐。矩阵 A 是稀疏的：A 中大多数元素都是 0，因为相对于所有可能的用户－艺术家组合，只有很少一部分组合会出现在数据中。算法将 A 分解为两个小矩阵 X 和 Y 的乘积。矩阵 X 和矩阵 Y 非常“瘦”，

因为 A 有很多行和列，但 X 和 Y 的行很多而列很少（列数用 k 表示）。这 k 个列就是潜在因素，用于解释数据中的交互关系。

由于 k 的值小，矩阵分解算法只能是某种近似，如图 3-1 所示。

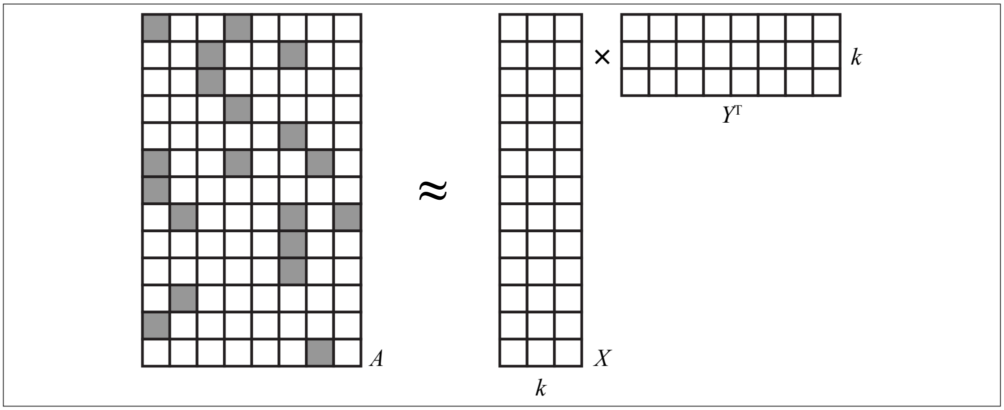


图 3-1：矩阵分解

矩阵分解算法有时称为矩阵补全（matrix completion）算法，因为原始矩阵 A 可能非常稀疏，但乘积 XY^T 是稠密的，即使该矩阵存在非零元素，非零元素的数量也非常少。因此模型只是对 A 的一种近似。原始 A 中大量元素是缺失的（元素值为 0），算法为这些缺失元素生成（补全）了一个值，从这个角度讲，我们可以把算法称为模型。

幸运的是，本例中的线性代数和我们的直觉很好地对应起来了。这种对应关系在这里是直接的，也是优雅的。两个矩阵分别有一行对应每个用户和每个艺术家。每行的值很少，只有 k 个。每个值代表了对应模型的一个隐含特征。因此行表示了用户和艺术家怎样关联到这 k 个隐含特征，而隐含特征可能就对应偏好或类别。于是问题就简化为用户 – 特征矩阵和特征 – 艺术家矩阵的乘积，该乘积的结果是对整个稠密的用户 – 艺术家相互关系矩阵的完整估计。该乘积可以理解成商品与其属性之间的一个映射，然后按用户属性进行加权。

不幸的是，$A = XY^T$ 通常根本没有确切的解，原因就是 X 和 Y 通常不够大（严格来讲就是矩阵的阶太小），无法完美表示 A。这其实也是件好事。A 只是所有可能出现的交互关系的一个微小样本。在某种程度上我们认为 A 是对基本事实的一次观察，它太稀疏，因此很难解释这个基本事实。但用少数几个因素（k 个）就能很好地解释这个基本事实。想象一下你正在玩拼图游戏，图案是一只猫。游戏最终答案很简单，就是一只猫。但当你手头上只有几块拼板时，就会很难描述眼前看到的图案。

XY^T 应该尽可能逼近 A，毕竟这是所有后续工作的基础，但它不能也不应该完全复制 A。然而同样不幸的是，想直接同时得到 X 和 Y 的最优解是不可能的。好消息是，如果 Y 已知，求 X 的最优解是非常容易的，反之亦然。但 X 和 Y 事先都是未知的。

幸好有算法可以帮助我们摆脱这种两难的境地，并且能找到一个还不错的解决方案。具体来说，求解 X 和 Y 时，本章使用交替最小二乘（Alternating Least Squares，ALS）算法。这类方法在 Netflix 竞赛期间流行起来，对此一些论文功不可没，比如“Collaborative Filtering for Implicit Feedback Datasets”和“Large-scale Parallel Collaborative Filtering for the Netflix Prize”。实际上 Spark MLlib 的 ALS 算法实现思想就来源于这两篇论文。

虽然 Y 是未知的，但我们可以把它初始化为随机行向量矩阵。接着运用简单的线性代数，就能在给定 A 和 Y 的条件下求出 X 的最优解。实际上，X 的第 i 行是 A 的第 i 行和 Y 的函数，因此可以很容易分开计算 X 的每一行。因为 X 的每一行可以分开计算，所以我们可以将其并行化，而并行化是大规模计算的一大优点。

$$A_i Y(Y^{\mathrm{T}} Y)^{-1} = X_i$$

要想两边精确相等是不可能的，因此实际的目标是最小化 $|A_i Y(Y^{\mathrm{T}} Y)^{-1} - X_i|$，或者最小化两个矩阵的平方误差。这就是算法名称中“最小二乘”的来由。这里给出方程式只是为了说明行向量计算方法，但实践中从来不会对矩阵求逆，我们会借助于 QR 分解（https://en.wikipedia.org/wiki/QR_decomposition）之类的方法，这种方法速度更快而且更直接。

同理，我们可以由 X 计算每个 Y_j。然后又可以由 Y 计算 X，这样反复下去。这就是算法名称中“交替”的来由。这里有一个小问题：Y 是“瞎编”的，并且是随机的。X 是最优化计算出来的，这没错，但给定的 Y 却是“假”的。好在，只要这个过程一直继续，X 和 Y 最终会收敛得到一个合适的结果。

将 ALS 算法用于隐性数据矩阵分解时，ALS 矩阵分解要稍微复杂一点儿。它不是直接分解输入矩阵 A，而是分解由 0 和 1 组成的矩阵 P，当 A 中元素为正时，P 中对应元素为 1，否则为 0。A 中的具体值后面会以权重的形式反映出来。本书不对其中细节做过多讨论，但我们有必要知道如何使用该算法。

最后，ALS 算法也可以利用输入数据是稀疏的这一特点。稀疏的输入数据、可以用简单的线性代数运算求最优解，以及数据本身可并行化，这 3 点使得算法在大规模数据上速度非常快。这也就是我们要把 ALS 算法作为本章主题的主要原因，同时也解释了为什么到目前为止 Spark MLlib 只有 ALS 一种推荐算法。

3.3 准备数据

首先，需要得到数据集的文件。将 3 个数据文件全部复制到 HDFS。本章假定文件放在 /user/ds/ 目录下，启动 `spark-shell`。注意本章的运算比简单的应用要占用更多的内存。如果运行在本地而不是在集群上，为了保证内存充足，在启动 `spark-shell` 时要指定参数 `--driver-memory 4g`。

构建模型的第一步是了解数据，对数据进行解析或转换，以便在 Spark 中做分析。

Spark MLlib 的 ALS 算法实现并不严格要求用户和产品的 ID 必须是数值型，不过当 ID 为 32 位非负整数时，效率会更高。使用 `Int` 表示 ID 是有好处的，但同时意味着 ID 不能超过 `Int` 的最大值（`Int.MaxValue`），即 2147483647。我们的数据集是否已经满足了这个要求？利用 `SparkSession` 的 `textFile` 方法，将数据文件转换成 `String` 类型的数据集：

```
val rawUserArtistData =
  spark.read.textFile("hdfs:///user/ds/user_artist_data.txt")

rawUserArtistData.take(5).foreach(println)

...
1000002 1 55
1000002 1000006 33
1000002 1000007 8
1000002 1000009 144
1000002 1000010 314
```

默认情况下，该数据集为每个 HDFS 块生成一个分区，将 HDFS 块大小设为典型的 128 MB 或 64 MB。由于 HDFS 文件大小为 400 MB，所以文件被拆为 3 个或 6 个分区。这通常没什么问题，但由于相比简单文本处理，ALS 这类机器学习算法要消耗更多的计算资源，因此减小数据块大小以增加分区个数会更好。减小数据块大小能使 Spark 处理任务时同时使用的处理器核数更多，因为每个核可以独立处理一个分区数据。可以在读取文本文件以后，接着调用一个 `.repartition(n)` 来指定一个不同于默认值的分区数，这样就可以将分区数设得大一些。比如，可以考虑将这个参数设为集群处理器总核数。

文件的每行包含一个用户 ID、一个艺术家 ID 和播放次数，用空格分隔。要计算用户 ID 的统计信息，可以用空格拆分每行，并将前两个值解析为整数，其结果在概念上可以看成 `Int` 类型的两个列：用户 ID 和艺术家 ID。将其转换为包含列 `user` 和 `artist` 的 DataFrame 是有意义的，因为这样就可以简单地计算出两列的最大值和最小值：

```
val userArtistDF = rawUserArtistData.map { line =>
  val Array(user, artist, _*) = line.split(' ') ❶
  (user.toInt, artist.toInt)
}.toDF("user", "artist")

userArtistDF.agg(
  min("user"), max("user"), min("artist"), max("artist")).show()

...
+---------+---------+-----------+-----------+
|min(user)|max(user)|min(artist)|max(artist)|
+---------+---------+-----------+-----------+
|       90|  2443548|          1|   10794401|
+---------+---------+-----------+-----------+
```

❶ 匹配并去掉剩余的标记。

最大的用户 ID 和艺术家 ID 分别是 2443548 和 10794401，而它们的最小值分别是 90 和 1，并没有出现负值。这些远比 2147483647 要小，所以在使用这些 ID 之前，没有必要进行额外的转换。

在这个例子后面的部分中，将会用到难以分辨的数字 ID 所对应的艺术家的名字，这些信息存储在 artist_data.txt 中。现在这个文件中包含了用制表符分割的艺术家 ID 和艺术家的名字。但是简单地把文件解析成二元组 (Int,String) 将会出错：

```
val rawArtistData = spark.read.textFile("hdfs:///user/ds/artist_data.txt")

rawArtistData.map { line =>
  val (id, name) = line.span(_ != '\t') ❶
  (id.toInt, name.trim)
}.count() ❷

...
java.lang.NumberFormatException: For input string: "Aya Hisakawa"
```

❶ 用第一个制表符分割行。

❷ 使用 .count 触发解析；这里会出错！

这里 span() 用第一个制表符将一行拆分成两部分，接着将第一部分解析为艺术家 ID，剩余部分作为艺术家的名字（去掉了空白的制表符）。文件里有少量行看起来是非法的：有些行没有制表符，有些行不小心加入了换行符。这些行会导致 NumberFormatException，它们不应该有输出结果。

然而，map() 函数要求对每个输入必须严格返回一个值，因此这里不能用这个函数。另一种可行的方法是用 filter() 方法删除那些无法解析的行，但这会重复解析逻辑。当需要将每个元素映射为零个、一个或更多结果时，我们应该使用 flatMap() 函数，因为它将每个输入对应的零个或多个结果组成的集合简单展开，然后放入到一个更大的数据集中。它可以和 Scala 集合一起使用，也可以和 Scala 的 Option 类一起使用。Option 代表一个值可以不存在，有点儿像只有 1 或 0 的一个简单集合，1 对应子类 Some，0 对应子类 None。因此在以下代码中，虽然 flatMap 中的函数本可以简单返回一个空 List，或一个只有一个元素的 List，但使用 Some 和 None 更合理，这种方法简单明了。

```
val artistByID = rawArtistData.flatMap { line =>
  val (id, name) = line.span(_ != '\t')
  if (name.isEmpty) {
    None
  } else {
    try {
      Some((id.toInt, name.trim))
    } catch {
```

```
      case _: NumberFormatException => None
    }
  }
}.toDF("id", "name")
```

这里返回一个 DataFrame，艺术家 ID 和名字分别对应列“id”和“name”。

artist_alias.txt 将拼写错误的艺术家 ID 或非标准的艺术家 ID 映射为艺术家的正规名字。其中每行有两个 ID，用制表符分隔。这个文件相对较小，有 200 000 个记录。有必要把它转成 Map 集合的形式，将“不良的”艺术家 ID 映射到“良好的”ID，而不是简单地把它作为包含艺术家 ID 二元组的数据集。这里又有一点小问题：由于某种原因有些行没有艺术家的第一个 ID。这些行将被过滤掉：

```
val rawArtistAlias = spark.read.textFile("hdfs:///user/ds/artist_alias.txt")
val artistAlias = rawArtistAlias.flatMap { line =>
  val Array(artist, alias) = line.split('\t')
  if (artist.isEmpty) {
    None
  } else {
    Some((artist.toInt, alias.toInt))
  }
}.collect().toMap

artistAlias.head
...
(1208690,1003926)
```

比如，第一条将 ID 1208690 映射为 1003926。接下来我们可以从包含艺术家名字的数据集中进行查找：

```
artistByID.filter($"id" isin (1208690, 1003926)).show()

...
+-------+----------------+
|     id|            name|
+-------+----------------+
|1208690|Collective Souls|
|1003926| Collective Soul|
+-------+----------------+
```

显然，这条记录将“Collective Souls”映射为“Collective Soul”。后者才是这支乐队正确的名称。

3.4 构建第一个模型

虽然现在数据集的形式完全符合 Spark MLlib 的 ALS 算法实现的要求，但我们还需要一个额外的转换。如果艺术家 ID 存在一个不同的正规 ID，我们要用别名数据集将所有的艺术

家 ID 转换成正规 ID。除此之外，只需要将输入的行解析成合适的列。可以定义一个辅助函数来做这件事情，以后还能重用它。

```
import org.apache.spark.sql._
import org.apache.spark.broadcast._

def buildCounts(
    rawUserArtistData: Dataset[String],
    bArtistAlias: Broadcast[Map[Int,Int]]): DataFrame = {
  rawUserArtistData.map { line =>
    val Array(userID, artistID, count) = line.split(' ').map(_.toInt)
    val finalArtistID =
      bArtistAlias.value.getOrElse(artistID, artistID) ❶
    (userID, finalArtistID, count)
  }.toDF("user", "artist", "count")
}

val bArtistAlias = spark.sparkContext.broadcast(artistAlias)

val trainData = buildCounts(rawUserArtistData, bArtistAlias)
trainData.cache()
```

❶ 如果艺术家存在别名，取得艺术家别名，否则取得原始名字。

虽然刚创建的 `artistAlias` 是驱动程序本地的一个 `Map`，我们仍然可以在 `map()` 函数中直接引用它。这是没问题的，因为 `artistAlias` 会随任务一起被自动复制。但是，它的体量可不小，要消耗大约 15 MB 内存，哪怕是序列化形式最少也得占用几兆字节。因为一个 JVM 中有许多任务，所以发送和存储如此多的副本太浪费了。

这时，我们可以为 `artistAlias` 创建一个广播变量，取名为 `bArtistAlias`。使用广播变量时，Spark 对集群中**每个** executor 只发送一个副本，并且在内存里也只保存一个副本。如果有几千个任务在 executor 上并行执行，使用广播变量能节省巨大的网络流量和内存。

广播变量

Spark 执行一个阶段（stage）时，会为待执行函数建立**闭包**，也就是该阶段所有任务所需信息的二进制形式。这个闭包包括驱动程序里函数引用的所有数据结构。Spark 把这个闭包发送到集群的每个 executor 上。

当许多任务需要访问同一个（不可变的）数据结构时，我们应该使用广播变量。它对任务闭包的常规处理进行扩展，使我们能够：

- 在每个 executor 上将数据缓存为原始的 Java 对象，这样就不用为每个任务执行反序列化；
- 在多个作业、阶段和任务之间缓存数据。

举个例子，考虑自然语言处理应用的场景，这里需要用到一本大型英语单词词典，以及一个接受一行字符和单词词典作为输入的评分函数。广播词典意味着对每个 executor 只要执行一次传输数据：

```
val dict: Seq[String] = ...
val bDict = spark.sparkContext.broadcast(dict)

def query(path: String) = {
  spark.read.textFile(path).map(score(_, bDict.value))
  ...
}
```

在连接一张大表和一张小表时，DataFrame 操作有时也会自动利用广播变量，这一点超出了本书的范围。在某些时候，广播一张小表性能更好，这称为**广播散列连接**（broadcast hash join）。

调用 `cache()` 以指示 Spark 在 DataFrame 计算好之后将其暂时存储在集群的内存里。这样是有益的，因为 ALS 算法是迭代的，通常情况下至少要访问该数据 10 次以上。如果不调用 `cache()`，那么每次要用到 DataFrame 时都需要从原始数据中重新计算。如图 3-2 所示，Spark UI 界面的 Storage 标签页显示了有多少 DataFrame 被缓存起来了，占用了多少内存。图中 DataFrame 占用了集群将近 120 MB 的内存。

Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
Memory Deserialized 1x Replicated	8	100%	120.3 MB	0.0 B

图 3-2：Spark UI 的 Storage 标签页，显示缓存 DataFrame 内存使用情况

注意，上面的 UI 中，Deserialized 标签实际上只与 RDD 相关，这里 Serialized 意味着数据是序列化成字节的形式存储在内存中的，而不是对象的形式。但是，像这样的 `Dataset` 和 `DataFrame` 实例，会在内存中分别执行它们自己“编码”的公共数据类型。

实际上，120 MB 可以说小得惊人。考虑到这里存储了大约 2400 万条播放记录，快速粗略估计一下可以发现，这意味着每个 user-artist-count 的组合平均仅消耗了 5 字节。然而，3 个 32 位的整型就能消耗 12 字节。这是 DataFrame 的优点之一。因为存储的数据类型是原始的 32 位整型，所以能在内存中内部优化数据的表示方法。如果换成原始基于 RDD API 的 ALS，要存储 2400 万个 `Rating` 对象，RDD 占用内存将超过 900 MB。

最后，我们构建模型：

```
import org.apache.spark.ml.recommendation._
import scala.util.Random
```

```
val model = new ALS().
    setSeed(Random.nextLong()). ❶
    setImplicitPrefs(true).
    setRank(10).
    setRegParam(0.01).
    setAlpha(1.0).
    setMaxIter(5).
    setUserCol("user").
    setItemCol("artist").
    setRatingCol("count").
    setPredictionCol("prediction").
    fit(trainData)
```

❶ 使用随机种子。

这样我们就构建了一个带有默认配置的 `ALSModel` 模型。这个操作可能要花费几分钟或者更长时间，具体时间取决于所用的集群。有些机器学习模型最终可能只有几个参数或系数，相比之下，我们这里使用的模型是巨大的。对于每个用户和产品，模型都包含一个有 10 个值的特征向量。在本章的示例中，总共有超过 170 万个特征向量。模型用两个不同的 DataFrame，它们分别表示“用户 – 特征”和“产品 – 特征”这两个大型矩阵。

你看到的结果会有些不同，原因是最终的模型取决于初始特征向量，而这些初始特征向量是随机选择的。然而，MLlib 的 ALS 模型和其他组件默认设置了固定的随机种子，每次都会做出相同的随机选择。这一点和其他库不一样，在默认情况下，一般库的随机元素通常不是固定的。所以，在这里和以后使用 MLlib 时，需要使用 `setSeed(Random.nextLong())` 设置一个真正的随机种子。

想看看某些特征向量，试试以下代码。它只显示一行，并且不截断特征向量显示宽度：

```
model.userFactors.show(1, truncate = false)

...
+---+----------------------------------------------- ...
|id |features                                        ...
+---+----------------------------------------------- ...
|90 |[-0.2738046, 0.03154172, 1.046261, -0.52314466, ...
+---+----------------------------------------------- ...
```

`ALS` 中的其他方法，如 `setAlpha`，会设置**超参数**，它们的值将影响模型的推荐质量，我们稍后再详细解释。更重要的是，首先要问：模型质量怎样？模型能给出好的推荐吗？

3.5　逐个检查推荐结果

应该看看模型给出的艺术家推荐直观上是否合理，我们检查一下用户播放过的艺术家，然后看看模型向用户推荐的艺术家。具体来看看用户 2093760 的例子。首先，我们观察他 /

她的播放记录来了解其品味。现在我们要提取该用户收听过的艺术家 ID 并打印他们的名字，这意味着先在输入数据中搜索该用户收听过的艺术家的 ID，然后用这些 ID 对艺术家集合进行过滤，这样我们就可以获取并按序打印这些艺术家的名字：

```
val userID = 2093760

val existingArtistIDs = trainData.
  filter($"user" === userID). ❶
  select("artist").as[Int].collect() ❷

artistByID.filter($"id" isin (existingArtistIDs:_*)).show() ❸


...
+-------+--------------+
|     id|          name|
+-------+--------------+
|   1180|    David Gray|
|    378|  Blackalicious|
|    813|    Jurassic 5|
|1255340|The Saw Doctors|
|    942|        Xzibit|
+-------+--------------+
```

❶ 找到用户 2093760 对应的行。

❷ 收集艺术家 ID 的整型集合。

❸ 过滤艺术家；:_* 变长参数语法。

用户播放过的艺术家既有大众流行音乐风格的也有嘻哈风格的。难道用户是 Jurassic 5 乐队的粉丝？记住这是 2005 年。顺便解释一下：Saw Doctors 是一支典型爱尔兰风格的摇滚乐队，在爱尔兰非常受欢迎。

不好的方面是，ALS 模型竟然没有提供直接计算用户最佳推荐的方法。用户最佳推荐的目的是评估用户对任意给定艺术家的偏好。Spark 2.2 加入了一个 `recommendAll` 方法来解决这个问题，但是在撰写本文的时候 Spark 2.2 还没有发布 [2]。`recommendAll` 方法可以用来给所有的艺术家打分，然后返回其中分值最高的。

```
def makeRecommendations(
    model: ALSModel,
    userID: Int,
    howMany: Int): DataFrame = {

  val toRecommend = model.itemFactors.
    select($"id".as("artist")).
    withColumn("user", lit(userID)) ❶
```

注 2：Spark 2.2 已于 2017 年 7 月 11 日正式发布。——译者注

```
  model.transform(toRecommend).
    select("artist", "prediction").
    orderBy($"prediction".desc).
    limit(howMany) ❷
}
```

❶ 选择所有艺术家 ID 与对应的目标用户 ID。

❷ 对所有艺术家评分，并返回其中分值最高的。

请注意，此方法不必过滤用户已经听过的艺术家的 ID。虽然这是很常见的需求，但并不是必需的，因为不过滤也不会影响我们最终的目标。

现在，做出推荐就很简单了，虽然采用这种方式计算它们需要一些时间。因此，它适用于批量评分，而不适用于实时评分。

```
val topRecommendations = makeRecommendations(model, userID, 5)
topRecommendations.show()

...
+-------+-----------+
| artist| prediction|
+-------+-----------+
|   2814|0.030201003|
|1300642|0.029290354|
|1001819|0.029130368|
|1007614|0.028773561|
|1037970|0.028646756|
+-------+-----------+
```

结果包含了一个艺术家 ID 和一个“预测”。虽然字段名称叫 `rating`，但其实不是估计的得分。对这类 ALS 算法，预测是一个 0~1 的模糊值，值越大，推荐质量越好。它不是概率，但可以把它理解成对 0/1 值的一个估计，0 表示用户不喜欢播放艺术家的歌曲，1 表示喜欢播放艺术家的歌曲。

得到所推荐艺术家的 ID 之后，就可以用类似的方法查到艺术家的名字：

```
val recommendedArtistIDs =
  topRecommendations.select("artist").as[Int].collect()

artistByID.filter($"id" isin (recommendedArtistIDs:_*)).show()

...
+-------+----------+
|     id|      name|
+-------+----------+
|   2814|   50 Cent|
|1007614|     Jay-Z|
|1037970|Kanye West|
|1001819|      2Pac|
|1300642|  The Game|
+-------+----------+
```

结果全部是嘻哈风格。我们一眼就能看出，这些推荐都不怎么样。虽然推荐的艺术家都受人欢迎，但好像并没有针对用户的收听习惯进行个性化。

3.6 评价推荐质量

当然，刚才只是对一个用户的推荐结果的一次主观评价。除了用户本人，其他任何人都很难对推荐的好坏给出定量描述。而且，想对推荐结果做人工评分，哪怕只评价一小部分结果，也是不切实际的。

我们假定用户会倾向于播放受人欢迎的艺术家的歌曲，而不会播放不受欢迎的艺术家的歌曲，这个假设是合理的。因此，用户的播放数据在一定程度上表示了“优秀的”和“糟糕的”艺术家推荐。这个假设虽然还有点儿问题，但是在没有其他数据的情况下，也只能这么做了。比如，170 万个艺术家，除了之前推荐的 5 个艺术家之外没有播放过的艺术家中，用户 2093760 很可能对其中某些感兴趣，所以不能说没有听过的艺术家都是“糟糕的”，都不能推荐。

如果根据好艺术家在推荐列表中排名应该靠前这个标准来评价推荐引擎，情况会怎样？推荐引擎这类的评分系统有几个指标，这个指标是其中之一。问题是如果将“好”的标准定义为“用户收听过艺术家”，那么推荐系统在输入中已经利用了这些信息。它可以简单把用户以前听过的艺术家作为最靠前的推荐结果返回，而这样就能得到最高的评价。然而这是无益的，因为推荐引擎的作用在于向用户推荐他从来没听过的艺术家。

为了使推荐变得有用，可以从数据集中拿出一些艺术家的播放数据放在一边，在整个 ALS 模型构建过程中并不使用这些数据。这些放在一边的数据中的艺术家可以作为每个用户的优秀推荐，但这些数据并没有喂给推荐引擎。让推荐引擎对模型中所有的产品进行评分，然后对比检查放在一边的艺术家的推荐排名情况。理想情况下，推荐引擎对这些艺术家的推荐排名应该最靠前或接近最靠前。

接着我们就可以计算推荐引擎的得分，方法是比较放在一边的艺术家推荐排名和整个数据集中的艺术家的推荐排名（在实践中，我们通常只比较一小部分，因为需要比较的艺术家组合可能非常多）。对比组合中放在一边的艺术家排名高的组合所占比例就是模型的得分。1.0 代表最好，0.0 代表最差，0.5 是随机给艺术家排名的模型的期望得分。

这个指标和一个信息检索概念直接相关，这个概念就是接受者操作特征（receiver operating characteristic，ROC，https://en.wikipedia.org/wiki/Receiver_operating_characteristic） 曲 线。上一段中的指标等于 ROC 曲线下区域的面积，称为 AUC（Area Under the Curve）。可以把 AUC 看成是随机选择的好推荐比随机选择的差推荐的排名高的概率。

AUC 指标也用于评价分类器。MLlib 的 `BinaryClassificationMetrics` 类实现了这个指标及相关方法。对于推荐引擎，为**每个用户**计算 AUC 并取其平均值，最后的结果指标稍有

不同，可称为“平均 AUC”。我们需要自己来实现，因为它不是在 Spark 中实现的。

其他和评分系统相关的评价指标在 `RankingMetrics` 类中实现。这些指标包括准确率、召回率和平均准确率（Mean Average Precision，MAP，https://en.wikipedia.org/wiki/Information_retrieval）。MAP 也常用，它更强调排在最前面的推荐的质量。但是，AUC 作为一种普遍和综合的测量整体模型输出质量的手段，是我们采用的。

事实上，取出一部分数据来选择模型并评估模型准确率是所有机器学习的通用做法。通常数据被分成三个子集：训练集、交叉验证（Cross-Validation，CV）集和测试集。在本章的初步示例中，我们只用了两个数据集：训练集和交叉验证集。这对于模型选择来说已经足够了。第 4 章会进一步讨论这个概念并介绍测试集。

3.7 计算AUC

平均 AUC 的具体实现请参考本书附带的源代码。代码实现比较复杂，请参考源代码的注释，这里我们就不重复说明了。该实现接受一个交叉验证集和一个预测函数，交叉验证集代表每个用户对应的“正面的”或“好的”艺术家。预测函数把每个包含“用户－艺术家”对的 DataFrame 转换为一个同时包含“用户－艺术家”和“预测”的 DataFrame，“预测”表示“用户”与“艺术家”之间关联的强度值，这个值越高，代表推荐的排名越高。

为了利用输入数据，我们需要把它分成训练集和验证集。训练集只用于训练 ALS 模型，验证集用于评估模型。这里我们将 90% 的数据用于训练，剩余的 10% 用于交叉验证：

```
def areaUnderCurve(
    positiveData: DataFrame,
    bAllArtistIDs: Broadcast[Array[Int]],
    predictFunction: (DataFrame => DataFrame)): Double = {
  ...
}

val allData = buildCounts(rawUserArtistData, bArtistAlias) ❶
val Array(trainData, cvData) = allData.randomSplit(Array(0.9, 0.1))
trainData.cache()
cvData.cache()

val allArtistIDs = allData.select("artist").as[Int].distinct().collect() ❷
val bAllArtistIDs = spark.sparkContext.broadcast(allArtistIDs)

val model = new ALS().
    setSeed(Random.nextLong()).
    setImplicitPrefs(true).
    setRank(10).setRegParam(0.01).setAlpha(1.0).setMaxIter(5).
    setUserCol("user").setItemCol("artist").
    setRatingCol("count").setPredictionCol("prediction").
    fit(trainData)
areaUnderCurve(cvData, bAllArtistIDs, model.transform)
```

❶ 注意这个函数已经在前文定义过了。

❷ 去重并收集给驱动程序。

注意：`areaUnderCurve()` 把一个**函数**作为它的第三个参数。这里传入的是 `ALSModel` 的 `transform`，很快我们会把它替换成其他方法。

结果约为 0.879。这个结果好吗？它肯定比随机推荐的 0.5 要好，并且接近最高分 1.0。一般 AUC 超过 0.9 是高分。

可以从数据集中选择另外的 90% 作为训练集，这样就可以多次进行模型评估。得到的 AUC 值的平均可能会更好地估计算法在数据集上的表现。实际中一个常用的做法是把数据集分成 *k* 个大小差不多的子集，用 *k*–1 个子集做训练，在剩下的一个子集上做评估。我们把这个过程重复 *k* 次，每次用一个不同的子集做评估。这种做法称为 *k* 折交叉验证（k-fold cross-validation，https://en.wikipedia.org/wiki/Cross-validation_(statistics)）算法。为了简便，我们在示例中并没有实现 *k* 折交叉验证技术。但 MLlib 的 `CrossValidator` API 在一定程度上提供了对这项技术的支持。这一验证用的 API 在第 4 章还会出现。

有必要把上述方法和一个更简单方法做一个基准比对。举个例子，考虑下面的推荐方法：向每个用户推荐播放最多的艺术家。这个策略一点儿都不个性化，但它很简单，也可能有效。定义这个简单预测函数并评估它的 AUC 得分：

```
def predictMostListened(train: DataFrame)(allData: DataFrame) = {

  val listenCounts = train.
    groupBy("artist").
    agg(sum("count").as("prediction")).
    select("artist", "prediction")

  allData.
    join(listenCounts, Seq("artist"), "left_outer").
    select("user", "artist", "prediction")
}

areaUnderCurve(cvData, bAllArtistIDs, predictMostListened(trainData))
```

这里再次显示了 Scala 语法的特别之处。这里函数定义看似有两个参数列表。调用函数并应用前两个参数得到了一个**偏应用函数**（partially applied function），这个函数本身又带一个参数（`allData`）并返回预测结果。`predictMostListened(sc, trainData)` 的返回结果是一个**函数**。

结果得分大约是 0.88。这意味着，对 AUC 这个指标，非个性化的推荐表现已经不错了。然而，我们想要的是得分更高，也就是更为“个性化”的推荐。显然这个模型还有待改进。还有没有可能做得更好呢？

3.8 选择超参数

到目前为止，我们并没有对给出的超参数值做任何说明。这些值不是由算法学习得到的，而是由调用者指定的。配置的超参数包括以下几个。

- `setRank(10)`

  模型的潜在因素的个数，即“用户－特征”和“产品－特征”矩阵的列数；一般来说，它也是矩阵的阶。

- `setMaxIter(5)`

  矩阵分解迭代的次数；迭代的次数越多，花费的时间越长，但分解的结果可能会更好。

- `setRegParam(0.01)`

  标准的过拟合参数，通常也称作 lambda；值越大越不容易产生过拟合，但值太大会降低分解的准确率。

- `setAlpha(1.0)`

  控制矩阵分解时，被观察到的“用户－产品”交互相对没被观察到的交互的权重。

可以把 `rank`、`regParam` 和 `alpha` 看作模型的**超参数**。（`maxIter` 更像是对分解过程使用的资源的一种约束。）这些值不会体现在 `ALSModel` 的内部矩阵中，这些矩阵只是**参数**，其值由算法选定。超参数则是构建过程本身的参数。

刚才列表中给出的超参数值不一定是最优的。如何选择好的超参数值在机器学习中是个普遍性问题。最基本的方法是尝试不同值的组合并对每个组合评估某个指标，然后挑选指标值最好的组合。

在下面的示例中，我们尝试了 8 种可能的组合：`rank = 5` 或 30，`regParam = 4.0` 或 0.0001，以及 `alpha = 1.0` 或 40.0。这些值当然也是猜的，但它们能够覆盖很大范围的参数值。各种组合的结果按 AUC 得分从高到底排序：

```
val evaluations =
  for (rank     <- Seq(5,  30);
       regParam <- Seq(4.0, 0.0001);
       alpha    <- Seq(1.0, 40.0)) ❶
    yield {
      val model = new ALS().
        setSeed(Random.nextLong()).
        setImplicitPrefs(true).
        setRank(rank).setRegParam(regParam).
        setAlpha(alpha).setMaxIter(20).
        setUserCol("user").setItemCol("artist").
        setRatingCol("count").setPredictionCol("prediction").
        fit(trainData)
```

```
        val auc = areaUnderCurve(cvData, bAllArtistIDs, model.transform)

        model.userFactors.unpersist() ❷
        model.itemFactors.unpersist()

        (auc, (rank, regParam, alpha))
      }

evaluations.sorted.reverse.foreach(println) ❸

...
(0.8928367485129145,(30,4.0,40.0))
(0.891835487024326,(30,1.0E-4,40.0))
(0.8912376926662007,(30,4.0,1.0))
(0.889240668173946,(5,4.0,40.0))
(0.8886268430389741,(5,4.0,1.0))
(0.8883278461068959,(5,1.0E-4,40.0))
(0.8825350012228627,(5,1.0E-4,1.0))
(0.8770527940660278,(30,1.0E-4,1.0))
```

❶ 可以理解为 3 层嵌套 `for` 循环。

❷ 立即释放模型占用的资源。

❸ 按第一个值（AUC）的降序排列并输出。

> 这里的 `for` 语法是 Scala 中写嵌套循环的一种方式，相当于一个 `alpha` 循环外面嵌套一个 `regParam` 循环，外面再嵌套一个 `rank` 循环。

虽然这些值的绝对差很小，但对于 AUC 值来说，仍然具有一定的意义。有意思的是，参数 `alpha` 取 40 的时候看起来总是比取 1 表现好（为了满足读者的好奇，顺便提一下，40 是前面提到的最初 ALS 论文的默认值之一）。这说明了模型在强调用户听过什么时的表现要比强调用户没听过什么时要好。

`regParam` 取较大的值看起来结果要稍微好一些。这表明模型有些受过拟合的影响，因此需要一个较大的 `regParam` 值以防止过度精确拟合每个用户的稀疏输入数据。第 4 章将进一步讨论过拟合。

正如预期的那样，对于这种体量的模型来说，5 个特征有点少，这个模型的表现要逊于使用 30 个用户品味特征的模型。正确的特征值个数实际上可能大于 30，而特征值个数太小时，无论是多少，区别都不大。

当然我们可以重复上述过程，试试不同的取值范围或试试更多值。这是超参数选择的一种暴力方式。但是在当今这个世界，这种简单粗暴的方式变得相对可行：集群常常有几 TB 内存，成百上千个核，Spark 之类的框架可以利用并行计算和内存来提高速度。

严格来说，理解超参数的含义其实不是必需的，但知道这些值的典型范围有助于找到一个合适的参数空间开始搜索，这个空间不宜太大，也不能太小。

我们目前使用的是较为原始的手动调参过程：设置超参数、构建模型、评估模型的三部曲。在第 4 章中学习更多 Spark ML API 以后，会发现一种更自动化的方式——管道（Pipeline）+ TrainValidationSplit。

3.9 产生推荐

现在用上一节中得到的最优超参数继续下面的工作，看看新模型对 ID 为 2093760 的用户给出什么样的推荐：

```
+-----------+
|       name|
+-----------+
|  [unknown]|
|The Beatles|
|     Eminem|
|         U2|
|  Green Day|
+-----------+
```

邮电

令人欣慰的是，现在给出的推荐要更符合这位用户的品味一些，其中大部分是流行摇滚，而非之前那样全都是嘻哈风格。推荐列表中 [unknown] 明显不是一个艺术家。查看原始数据集会发现 [unknown] 出现了 429 447 次，几乎可以排到前 100 了。[unknown] 是没有艺术家信息的播放记录的一个默认值，可能是某个客户端提供的。这个信息是没有用的，下次我们应该在开始的时候把它扔掉。这又再次说明，数据科学实践往往是迭代式的，每个阶段我们对数据都有新发现。

这个模型可以对所有用户产生推荐。它可以用于批处理，批处理每隔一个小时或更短的时间为所有用户重算模型和推荐结果，具体时间间隔取决于数据规模和集群速度。

但是目前 Spark MLlib 的 ALS 实现并不支持向所有用户给出推荐。该实现可以每次对一个用户进行推荐，这样每一次都会启动一个短的几秒钟的分布式作业。这适合对小用户群体快速重算推荐。下面对数据中的 100 个用户进行推荐并打印结果：

```
val someUsers = allData.select("user").as[Int].distinct().take(100) ❶
val someRecommendations =
  someUsers.map(userID => (userID, makeRecommendations(model, userID, 5))) ❷
someRecommendations.foreach { case (userID, recsDF) =>
  val recommendedArtists = recsDF.select("artist").as[Int].collect()
  println(s"$userID -> ${recommendedArtists.mkString(", ")}") ❸
}

...
```

```
1000190 -> 6694932, 435, 1005820, 58, 1244362
1001043 -> 1854, 4267, 1006016, 4468, 1274
1001129 -> 234, 1411, 1307, 189, 121
...
```

❶ 把 100 个（不同的）用户复制到驱动程序端。

❷ `map()` 在这里是一个本地 Scala 运算。

❸ `mkString` 用分隔符把集合中的元素连接成一个字符串。

现在只是把推荐结果打印出来。结果也可以写到外部存储上，比如 HBase（https://hbase.apache.org/）上，这样可以在运行时利用 HBase 提供快速查询。

有意思的是，整个流程也可能用于向**艺术家**推荐**用户**。这可用于回答类似这样的问题："艺术家 *X* 的新专辑，哪 100 个用户可能最感兴趣？"在向艺术家推荐用户时，只需要在解析输入的时候对换用户和艺术家字段就可以了。

```
rawArtistData.map { line =>
  val (id, name) = line.span(_ != '\t')
  (name.trim, id.int)
}
```

3.10 小结

显然我们可以花多点儿时间来对模型参数进行调优，找出输入数据中的异常情况，比如说有的艺术家名字为 `[unknown]`，并修复这些问题。举个例子来说，对播放次数进行快速分析就会发现，ID 为 2064012 的用户播放 ID 为 4468 的艺术家高达 439 771 次，这太让人吃惊了。ID 为 4468 的艺术家是杰出的独立金属乐队 System of a Down，之前的推荐中出现过。假定每首歌曲长度为 4 分钟，如果播放"Chop Suey!"和"B.Y.O.B."这样的热门歌曲，33 年也完成不了。因为乐队 1998 年才开始录制唱片，即使同时播放 4~5 首单曲也要 7 年，所以这肯定是垃圾数据、数据错误或者某类实际数据问题，这些问题生产系统必须要解决。

ALS 不是唯一的推荐引擎算法。目前它是 Spark MLlib 唯一支持的算法。但是，对于非隐含数据，MLlib 也支持一种 ALS 的变体，它的用法和 ALS 是一样的，不同之处在于 ALS 使用 `setImplicitPrefs(false)` 配置。它适用于给出评分数据而不是次数数据。比如，如果数据集是用户对艺术家的打分，取值范围是 1~5，那么用这种变体就很合适。推荐方法 `ALSModel.transform` 返回结果列 `prediction`，它才是模型估算出的评分。这种情况下，简单的均方根误差（root mean squared error，RMSE）度量标准就能够评价推荐算法了。

今后 Spark MLlib 或其他库可能支持其他推荐算法。

在生产环境下，推荐引擎需要实时给出推荐，因为它们常用于电商网站。在这类环境中，

客户浏览商品页面时需要推荐引擎频繁给出推荐。像前面提到的那样，预先计算并把得到的推荐结果存到一个 NoSQL 存储中，在大规模情况下不失为一种合理的策略。这种做法的一个缺点是需要对所有可能提供快速推荐的用户进行预计算，但这些用户可能是任何一个用户。举例来讲，即使 100 万个用户中每天只有 10 000 个访问网站，我们也要为 100 万个用户做预计算，其中 99% 的工作是浪费的。

如果能随时按需计算出推荐结果就更好了。虽然我们可以用 `ALSModel` 为单个用户计算推荐结果，它却还是一个分布式运算，需要花费几秒钟。因为 `ALSModel` 非常大，所以实际上是个分布式数据集。其他模型情况有些不同，它们可以更快地给出评分。Oryx 2（https://github.com/OryxProject/oryx）之类的项目试图实现实时按需推荐，其底层用 MLlib 之类的库，但用高效的方式访问内存中的模型数据。

第 4 章

用决策树算法预测森林植被

作者：肖恩·欧文

> 预测是非常困难的，更别提预测未来。
>
> ——Niels Bohr

19 世纪末，英国科学家弗兰西斯·高尔顿爵士（Francis Galton，达尔文的表兄）忙于测量豌豆的大小和人类的身高来寻找规律。他发现大豌豆的子代一般会大于子代豌豆的平均大小（人类也如此）。这没有什么好奇怪的。但是，大豌豆的子代通常比它们的父代要小。对人类而言，身高 7 英尺的篮球运动员的孩子很可能比一般人要高，但是身高达到 7 英尺的可能性却较小。

高尔顿这个研究的另一个成果是，他通过绘制子代大小与父代大小的关系图，发现二者之间存在近似的线性关系。父代豌豆大，子代豌豆也大，但要略小于父代豌豆；父代豌豆小，子代豌豆也小，但要略大于父代豌豆。因此曲线斜率为正但小于 1，高尔顿把这种现象称为**趋均数回归**（regression to the mean），此名称被沿用至今。

依我看，这条线就是早期的预测模型，尽管当时还没有人这么提出。这条线把两个值关联在一起，意味着知道了一个值就大体知道了另一个值。如果知道一颗新豌豆的大小，根据这种关联关系，我们就能更准确估计其后代的大小，这样做出的估计要比简单假设“子代”的大小接近于“父代或同类”更准确。

4.1 回归简介

经过随后一百多年统计学的发展，随着现代机器学习和数据科学的出现，我们依旧把从“某些值”预测“另外某个值”的思想称为**回归**（https://en.wikipedia.org/wiki/Regression_analysis），即使它已经和“趋均数回归”没有任何关系，也跟“向后移动”不沾边。回归技术和分类（https://en.wikipedia.org/wiki/Statistical_classification）技术关系紧密。通常来讲，**回归**是预测一个数值型数量（numeric quantity），比如大小、收入和温度，而**分类**则指预测标号（label）或类别（category），比如判断邮件是否为“垃圾邮件”，拼图游戏的图案是否为“猫”。

将回归和分类联系在一起是因为两者都可以通过一个（或更多）值预测另一个（或更多）值。为了能够做出预测，两者都需要从一组输入和输出中学习预测规则。在学习过程中，需要告诉它们问题以及问题的答案。因此，它们都属于所谓的**监督学习**（https://en.wikipedia.org/wiki/Supervised_learning）。

分类和回归是分析预测中最古老的话题，也是被研究得最多的一类问题。分析用的程序包（package）和程序库（library）中的多数算法，都属于分类和回归技术，比如支持向量机、logistic 回归、朴素贝叶斯算法、神经网络和深度学习。第 3 章介绍的推荐系统，相对容易解释，但那也是机器学习领域中一个相对比较新和比较独立的子课题。

本章将重点关注**决策树算法**（https://en.wikipedia.org/wiki/Decision_tree）和它的扩展**随机决策森林算法**（https://en.wikipedia.org/wiki/Random_forest），这两个算法灵活且应用广泛，既可用于分类问题，也可用于回归问题。更让人兴奋的是，它们可以帮助我们预测未来，至少是预测我们尚不肯定的事情。比如说，根据线上行为来预测购买汽车的概率，根据用词预测邮件是否是垃圾邮件，根据地理位置和土壤的化学成分预测哪块耕地的产量可能更高。

4.2 向量和特征

为了便于解释本章选择的数据集和要着重介绍的算法，以及为了便于下一步介绍回归和分类的原理，有必要先对描述输入和输出的术语做个简短定义。

我们想根据今天的天气预测明天的气温。这个没有问题，但“今天的天气”是个宽泛的概念，并且在用于机器学习算法之前需要对它进行结构化。

“今天的天气”中某些“**特征**”也许可以用来预测明天的气温，比如：

- 今天的最低气温
- 今天的最高气温
- 今天的平均湿度

- 今天是多云、有雨还是晴朗
- 今天有几家天气预报站预报明天有寒流

这些特征有时也被称为**维度**、**预测指标**或简单地称为**变量**。以上每个特征都可以被量化。比如气温高低可以用“摄氏度”度量，湿度可以用 0~1 范围内的小数来度量，天气类型可以用“多云”“有雨”和“晴朗”来标示。天气预报的个数则是个整数。因此今天的天气可以简化为一个值列表：`13.1，19.0，0.73，多云 ,1`。

这 5 个特征值按顺序排列，就是所谓的**特征向量**，它可用于描述每天的天气。这种用法与线性代数里的“向量”比较类似，但稍微不同的是，这里我们所说的向量值可以包含非数值，有些值甚至可以为空。

这些特征值的类型各有不同。前两个特征用“摄氏度”来计量，第三个特征则是个不带单位的小数。第四个根本就不是数值，第五个是一个非负整数。

为了便于讨论，本书将特征只分为两大类：**类别型特征**（categorical feature）和**数值型特征**（numeric feature）。这里的数值型特征是指可以用数值进行量化的特征，并且对这些特征排序是有意义的。比如，今天最高气温为 23 摄氏度，它比昨天最高气温 22 摄氏度要高，这种说法是有意义的。除天气类型外，前面提到的所有特征都是数值型特征。“晴朗”这类词语不是数字，没有大小顺序可言。“多云比晴朗大”这种说法是没有意义的。天气类型就属于类别型特征，类别型特征只能在几个离散值中取一个。

4.3　样本训练

为了进行预测，学习算法需要在大量数据上进行训练。这需要从历史数据中获取大量的数据输入和相应的已知的正确的数据输出。比如，在示例中我们会向学习算法给出如下样本数据：某一天，气温 12~16 摄氏度，湿度 10%，晴朗，没有寒流预报，第二天最高气温为 17.2 摄氏度。如果这样的样本数量足够多，学习算法就可能学会以一定准确率来预测第二天最高气温。

特征向量为学习算法的输入提供了一种结构化的方式（如输入：`12.5,15.5,0.10,晴朗 ,0`）。预测的输出（即**目标**）也被称为一个特征，它是一个数值型特征：`17.2`。通常我们把目标作为特征向量的一个附加特征。因此可以把整个训练样本列示为：`12.5,15.5,0.10,晴朗 ,0,17.2`。所有这些样本的集合称为**训练集**。

读者要注意，回归和分类的区别在于：回归问题的目标为数值型特征，而分类问题的目标为类别型特征。并不是所有的回归或分类算法都能够处理类别型特征或类别型目标，有些算法只能处理数值型特征。

4.4 决策树和决策森林

事实证明，**决策树**算法家族能自然地处理类别型和数值型特征。单棵决策树可以并行构建，许多树也可以同时并行构建。它们对数据中的离群点（outlier）具有稳健性（robust），这意味着一些极端或可能错误的数据点根本不会对预测产生影响。算法可以接受不同类型和不同取值范围的数据，不需要将数据转化成同一类型，或是将数据规范化到特定的值域中。数据类型和数据取值范围不相同的问题在第 5 章会再次涉及。

决策树可以推广为更强大的**决策森林**（random decision forest）算法。由于决策森林算法的灵活性，我们认为有必要在本章介绍它。本章将会把 Spark MLlib 的 `DecisionTree` 和 `RandomForest` 算法实现应用到一个数据集上。

基于决策树的算法还有一个优点，那就是理解和推理起来相对直观。实际上，我们在日常生活中大概都无意间用过决策树所体现的推理方法。举个例子，早上我正要坐下来喝杯加奶咖啡，在打定主意在咖啡中加牛奶之前，我会预测：牛奶有没有变质呢？我不确定牛奶是否变质。于是我会检查一下牛奶的建议食用期。如果建议食用期没过，我会预测牛奶没有变质。如果已经过了建议食用期，但没有超过 3 天，我会猜测牛奶还没变质。如果超过建议食用期 3 天以上，我会闻一下。如果牛奶的气味有点儿异常，我会预测牛奶已经变质了，否则就没变质。

这一系列的“是 / 否”判断就是决策树算法所体现的预测过程。每个判断会走到两个分支中的一个，非此即彼，如图 4-1 所示。

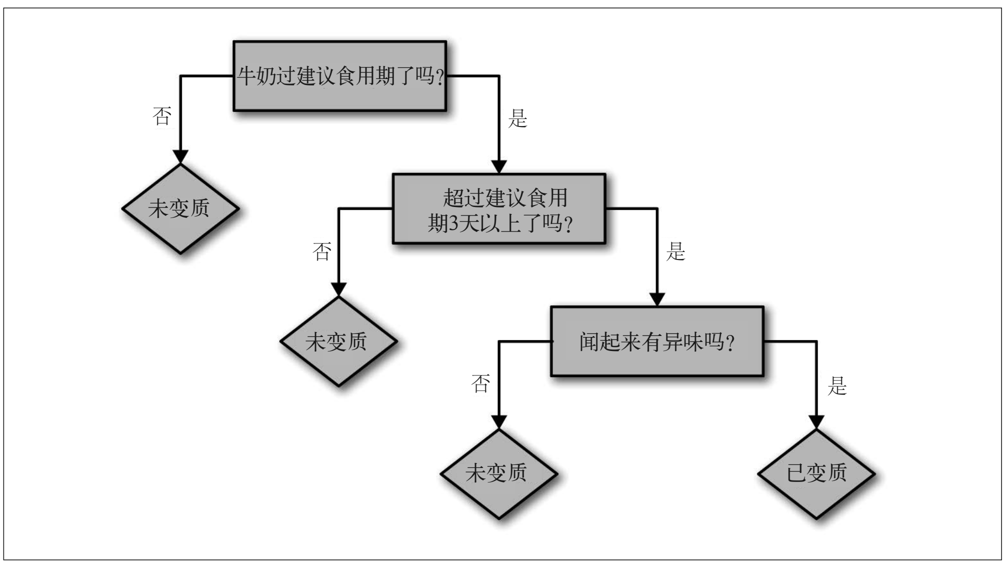


图 4-1：决策树——牛奶变质了吗

前面提到的规则是我在大学期间学会的，很直观。这些规则不但简单，而且能有效地帮我区分牛奶是否已经变质了。这些都是一棵好的决策树的特点。

上面是一棵简化的决策树，构造过程非常灵活。为了更详细地说明决策树，我们来看看另外一个例子。在一家新奇的宠物店，一个机器人正忙于工作。在宠物店开门营业之前，它要学习什么动物适合成为孩子们的宠物。在开始之前，店长给出了 9 种宠物，其中有的适合做宠物，有的不适合。经过一番检查，机器人把这些信息编写成一张表格，如表 4-1 所示。

表4-1：新奇宠物店的“特征向量”

名　字	重量（公斤）	腿　数	颜　色	适合做宠物吗
Fido	20.5	4	棕色	是
Mr. Slither	3.1	0	绿色	否
Nemo	0.2	0	棕黄色	是
Dumbo	1390.8	4	灰色	否
Kitty	12.1	4	灰色	是
Jim	150.9	2	棕黄色	否
Millie	0.1	100	棕色	否
McPigeon	1.0	2	灰色	否
Spot	10.0	4	棕色	是

虽然数据中已经给定了名字，但名字并不能作为一个特征。我们没有道理认为名字本身具有预测性：就机器人所知，“Felix”可能是只猫的名字，也可能是只有毒的狼蛛。因此，这里所剩的特征有：两个数值型特征（重量、腿数），一个类别型特征（颜色），这 3 个特征用来预测一个类别型目标（是否适合做小孩的宠物）。

机器人可能会用一个简单的决策树拟合训练数据集，这棵决策树只根据“重量”做决策，如图 4-2 所示。

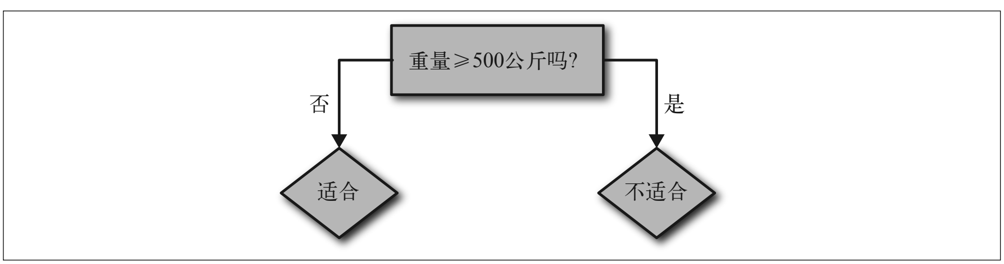


图 4-2：机器人的第一棵决策树

决策树逻辑很好理解，而且有一定的道理：500 公斤的动物肯定不适合做宠物。这条规则能对 9 个样本中的 5 个做出正确预测。快速看一下训练数据，我们就能发现把重量的阈值

降低为 100 公斤能够改进决策规则。这样我们就能正确预测 6 个样本。现在重的动物都能预测正确了，但轻的动物只能部分预测正确。

因此，为了进一步提高对体重小于 100 公斤的动物的预测准确率，机器人进行了第二次决策。如果能找到一个特征，通过这个特征将错误的“是”预测纠正为“否”预测，那就太好了。比如，训练集中有一种小型的绿色动物，看上去有点儿像条蛇，不适合做宠物，机器人就可以根据颜色对此做出正确判断，如图 4-3 所示。

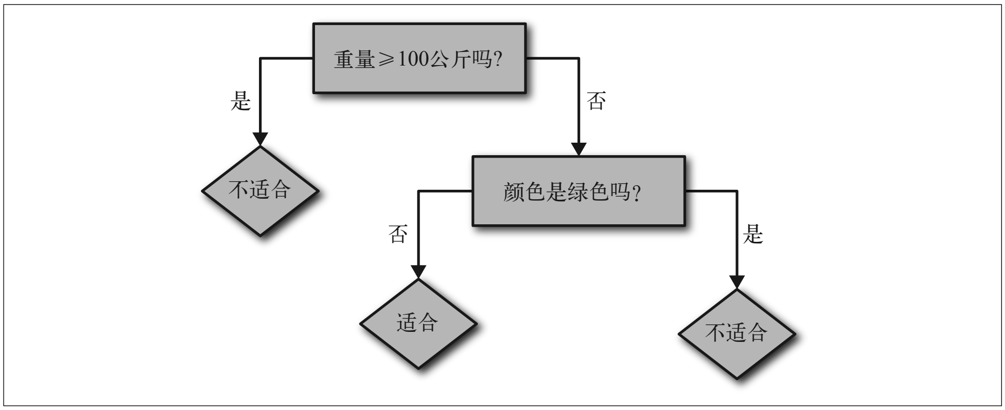


图 4-3：机器人的第二棵决策树

现在 9 个样本中有 7 个可以正确预测。当然，可以一直增加决策规则，直到对所有 9 个样本都能全部正确地做出预测。但这样得出的决策树很可能不合理，如果翻译成常用语，决策树可能是：“如果动物的重量小于 100 公斤，颜色是棕色而不是绿色，并且腿的数量少于 10，那么它适合做宠物。”虽然能完美拟合给定样本，但这样的决策树不能预测出棕色、有 4 条腿的小型狼獾不合适做宠物。看来，为避免这种被称为**过度拟合**的现象，还是需要继续改进啊。

不过到目前为止，对决策树算法的介绍已经足够我们在 Spark 中使用决策树算法了。本章接下来的内容将描述怎样选择决策规则，何时停止决策过程，以及怎样通过创建决策森林来提高准确率。

4.5 Covtype数据集

本章用到的数据集是著名的 Covtype 数据集，该数据集可以在线下载（https://archive.ics.uci.edu/ml/machine-learning-databases/covtype），包含一个 CSV 格式的压缩数据文件 covtype.data.gz，附带一个描述数据文件的信息文件 covtype.info。

该数据集记录了美国科罗拉多州不同地块的森林植被类型（也就是现实中的森林，这仅仅是巧合），每个样本包含了描述每块土地的若干特征，包括海拔、坡度、到水源的距离、遮阳情况和土壤类型，并且给出了地块对应的已知森林植被类型。我们需要总共 54 个特

征中的其余各项来预测森林植被类型。

人们已经用该数据集进行了研究，甚至在 Kaggle 大赛（https://www.kaggle.com/c/forest-cover-type-prediction）中也用过它。本章之所以研究这个数据集，原因在于它不但包含了数值型特征，还包含了类别型特征。该数据集有 581 012 个样本，虽然还称不上大数据，但作为一个范例来已经足够大，而且也能够反映出大数据上的一些问题。

4.6 准备数据

幸运的是，数据已经是简单的 CSV 格式了。在开始用 Spark 的 MLlib 之前，我们不需要进行大量的清洗或其他准备工作。后面我们将介绍如何对数据做一些转换，但现在该数据集可以直接开始用。

解压 covtype.data 文件并复制到 HDFS。本章我们假定文件放在 /user/ds/ 目录下。请启动 `spark-shell`。你可能会再次发现，由于创建决策森林是资源密集型的任务，给 `spark-shell` 提供足够大的内存是十分有帮助的。如果内存充足的话，可以通过 `spark-shell` 指定 `--driver-memory 8g` 或差不多大小的内存量。

CSV 文件基本上是表格数据，组织成许多行，每行又包含多个列。文件首行有时是列名，但此处首行并不是列名，列名在随附的文件 covtype.info 中。从概念上说，CSV 文件的每列还会有一个类型，比如数字类型、字符串类型，但是 CSV 文件并没有指明列的类型。

很自然地，我们把该数据解析成 DataFrame，因为 DataFrame 就是 Spark 针对表格数据的抽象，它有定义好的模式，包括列名和列类型。事实上，Spark 内置了读取 CSV 数据的功能：

```
val dataWithoutHeader = spark.read.
  option("inferSchema", true).
  option("header", false).
  csv("hdfs:///user/ds/covtype.data")

...
org.apache.spark.sql.DataFrame = [_c0: int, _c1: int ... 53 more fields]
```

这段代码读取 CSV 输入，但并没有把第一行解析为列名。通过检查数据来推断每列的类型，代码正确推断出所有列都是数值类型，更精确地说是整数。但不幸的是，代码只能把列名依次指定为 `_c0`、`_c1`，等等。

检查一下列名，可以清楚地看到，有些特征确实是数值型。“高度”以米为单位，“坡度”以度为单位。但是“荒野区域”有些不同，因为它横跨 4 列，每列要么为 0，要么为 1。实际上荒野区域是一个类别型特征，而非数值型。

这 4 列其实是 one-hot（https://en.wikipedia.org/wiki/One-hot）或 1-of-*n* 编码。在这种编码中，一个有 *N* 个不同取值的类别型特征可以变成 *N* 个数值型特征，变换后的每个数值型特

征的取值为 0 或 1。在这 N 个特征中，有且只有一个取值为 1，其他特征取值都为 0。比如，类别型特征“天气”可能的取值有“多云”“有雨”或“晴朗”。在 1-of-n 编码中，它就变成了 3 个数值型特征：多云用 `1,0,0` 表示，有雨用 `0,1,0` 表示，晴朗用 `0,0,1` 表示。可以为这 3 个数值型特征分别取名：`is_cloudy`、`is_rainy` 和 `is_clear`。同样，还有 40 列也是同一个类别型特征 `Soil_Type`。

这并不是将分类特性编码为数值的唯一方法。另一种可能的编码方式是为类别型特征的每个可能取值分配一个不同数值，比如多云 1.0、有雨 2.0 等。目标“Cover_Type”本身也是类别型值，用 1~7 编码。

> 在编码过程中，将类别型特征当成数值型特征时要小心。类别型特征值原本是没有大小顺序可言的，但被编码为数值之后，它们就“显得”有大小顺序了。被编码后的特征若被视为数值，算法在一定程度上会假定有雨比多云大，而且大两倍，这样就可能导致不合理的结果。当然，只要算法不把数字编码当作数值来用也没什么问题。

于是在 Covtype 数据集中，我们看到它同时使用了前面提到的类别型变量的两种编码方法。如果对类别型特征不用这两种编码方式，而是直接使用类似“Rawah Wilderness Area”这样的值，有可能会更简单更直接。这可能是历史的产物：Covtype 数据集在 1998 年发布。基于性能方面的考虑，或者为了满足当时处理工具要求的格式（当时工具更多是面向回归问题的），数据集往往用这样的方式进行编码。

无论如何，在处理 DataFrame 之前，将列名添加到其中是有帮助的，可以让它使用起来更方便：

```
val colNames = Seq(
    "Elevation", "Aspect", "Slope",
    "Horizontal_Distance_To_Hydrology", "Vertical_Distance_To_Hydrology",
    "Horizontal_Distance_To_Roadways",
    "Hillshade_9am", "Hillshade_Noon", "Hillshade_3pm",
    "Horizontal_Distance_To_Fire_Points"
  ) ++ ( ❶
    (0 until 4).map(i => s"Wilderness_Area_$i")
  ) ++ (
    (0 until 40).map(i => s"Soil_Type_$i")
  ) ++ Seq("Cover_Type")

val data = dataWithoutHeader.toDF(colNames:_*).
  withColumn("Cover_Type", $"Cover_Type".cast("double"))

data.head
...
org.apache.spark.sql.Row = [2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,...
```

❶ ++ 操作符可以连接两个集合。

“wilderness-”和“soil-related”列分别被命名为“Wilderness_Area_0”和“Soil_Type_0”。使用一些 Scala 代码可以生成这 44 个名字，不需要手动输入。最后，目标列“Cover_Type”被预先转换成双精度值，在 Spark MLlib 所有 API 中，目标列通常都被视为双精度浮点数类型而不是整型。这种转换是对用户透明的。

你可以调用 `data.show()` 来查看数据集的某些行，但是显示行非常宽，因此很难阅读。`data.head` 可以显示原始的 `Row` 对象，在这种情况下使用 `data.head` 具有更好的可读性。

4.7 第一棵决策树

第 3 章从所给数据中快速地建立了一个推荐模型。大家可以借助一些音乐知识凭感觉就能对这个推荐引擎做一些判断：只要对比看看用户的收听习惯和引擎推荐的艺术家，就能大概知道推荐引擎给出的推荐还不错。但在这里，这样做却是不可能的。我们既不知道怎样用 54 个特征来描述科罗拉多州的一个从未见过的地块，也不知道如何预测这种地块上的森林植被类型。

相反，我们可以直接从数据集中取出部分数据，用以评估所得到的模型。之前，为了评价保留的收听数据和模型预测之间的一致性，我们采用 AUC 指标。这里我们采用同样的原理，不过评价指标改为**准确率**（accuracy）指标。大部分数据（90%）会再次用作训练集，后面我们将看到该训练集的子集用于交叉验证，这个子集就是交叉验证集。剩下 10% 的数据实际上是第三个子集，会保留出来成为一个数量合理的测试集。

```
val Array(trainData, testData) = data.randomSplit(Array(0.9, 0.1))
trainData.cache()
testData.cache()
```

该数据集在用于 Spark MLlib 分类器之前需要做一些额外的预处理。输入 DataFrame 包含许多列，每列对应一个特征，可以用来预测目标列。Spark MLlib 要求将所有输入合并成一列，该列的值是一个向量。这个类是线性代数中向量的一个抽象，仅包含一些数字。对于大多数的意图和目的，向量就像是一个简单的双精度浮点数数组。当然，有些输入特征在概念上是类别型的，即使在输入中它们都是用数值表示的。现在先忽略这一点，稍后再讨论。

幸运的是，`VectorAssembler` 类可以完成这项工作：

```
import org.apache.spark.ml.feature.VectorAssembler

val inputCols = trainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val assembledTrainData = assembler.transform(trainData)
assembledTrainData.select("featureVector").show(truncate = false)
```

```
...
+-------------------------------------------------------------- ...
|featureVector                                                  ...
+-------------------------------------------------------------- ...
|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1863.0,37.0,17.0,120.0,18.0,90.0,2 ...
|(54,[0,1,2,5,6,7,8,9,13,18],[1874.0,18.0,14.0,90.0,208.0,209.0,135. ...
|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1879.0,28.0,19.0,30.0,12.0,95.0,20 ...
...
```

它的关键参数是要组合成特征向量的那些列，以及包含特征向量的新列的名称。这里**除了**目标列**以外**，所有其他列都作为输入特征，因此产生的 DataFrame 有一个新的"featureVector"列，如上所示。

输出看起来不是很像一串数字，这是因为它显示的是向量的原始表示，也就是 `SparseVector` 的实例，这样做可以节省存储空间。由于这 54 个值中的大多数值都是 0，它仅存储非零值及其索引。这个细节对分类来说无关紧要。

`VectorAssembler` 是当前 Spark MLlib"管道"API 中的一个 `Transformer` 示例。`VectorAssembler` 可以将一个 DataFrame 转换成另外一个 DataFrame，并且可以和其他 `Transformer` 组合成一个管道。在本章后面，这些转换操作将连接成一个真正的管道。此处我们直接调用转换操作，这对我们要构建的第一个决策树分类模型已经足够了。

```
import org.apache.spark.ml.classification.DecisionTreeClassifier
import scala.util.Random

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()). ❶
  setLabelCol("Cover_Type").
  setFeaturesCol("featureVector").
  setPredictionCol("prediction")

val model = classifier.fit(assembledTrainData)
println(model.toDebugString)

...
DecisionTreeClassificationModel (uid=dtc_29cfe1281b30) of depth 5 with 63 nodes
  If (feature 0 <= 3039.0)
   If (feature 0 <= 2555.0)
    If (feature 10 <= 0.0)
     If (feature 0 <= 2453.0)
      If (feature 3 <= 0.0)
       Predict: 4.0
      Else (feature 3 > 0.0)
       Predict: 3.0
       ...
```

❶ 使用随机种子。

同样，分类器的主要配置包括列名：包含输入特征向量的列和包含被预测特征值的列。

因为后面我们将把模型用于预测新的目标值，所以将模型命名为存储预测值的列的名称。

依据上面模型表示方式的输出信息，我们可以发现模型的一些树结构。它由一系列针对特征的嵌套决策组成，这些决策将特征值与阈值相比较。（很不幸，由于历史原因，这里的这些特性只用数字来引用，而不是按名称引用。）

构建决策树的过程中，决策树能够评估输入特征的重要性。也就是说，它们可以评估每个输入特征对做出正确预测的贡献值。从模型中很容易获得这个信息。

```
model.featureImportances.toArray.zip(inputCols).
  sorted.reverse.foreach(println)

...
(0.7931809106979147,Elevation)
(0.050122380231328235,Horizontal_Distance_To_Hydrology)
(0.030609364695664505,Wilderness_Area_0)
(0.03052094489457567,Soil_Type_3)
(0.026170212644908816,Hillshade_Noon)
(0.024374024564392027,Soil_Type_1)
(0.0167000614217678 7,Soil_Type_31)
(0.012596990926899494,Horizontal_Distance_To_Roadways)
(0.011205482194428473,Wilderness_Area_2)
(0.002419427115249023 5,Hillshade_3pm)
(0.0018551637821715788,Horizontal_Distance_To_Fire_Points)
(2.450368306995527E-4,Soil_Type_8)
(0.0,Wilderness_Area_3)
...
```

这样我们就把列名及其重要性（越高越好）关联成二元组，并按照重要性从高到低排列输出。Elevation 似乎是绝对重要的特征；其他的大多数特征在预测封面类型时几乎没有任何作用！

返回的结果 DecisionTreeClassificationModel 本身就是一个转换器，因为它可以将一个包含特征向量的 DataFrame 转换成一个包含特征向量及其预测结果的 DataFrame。例如，看看模型在训练数据上的预测结果是有帮助的，可以比较一下模型预测值与正确的覆盖类型。

```
val predictions = model.transform(assembledTrainData)
predictions.select("Cover_Type", "prediction", "probability").
  show(truncate = false)

...
+----------+----------+------------------------------------------------ ...
|Cover_Type|prediction|probability                                      ...
+----------+----------+------------------------------------------------ ...
|6.0       |3.0       |[0.0,0.0,0.03421818804589827,0.6318547696523378, ...
|6.0       |4.0       |[0.0,0.0,0.043440860215053764,0.283870967741935, ...
|6.0       |3.0       |[0.0,0.0,0.03421818804589827,0.6318547696523378, ...
|6.0       |3.0       |[0.0,0.0,0.03421818804589827,0.6318547696523378, ...
...
```

有趣的是，输出还包含了一个“probability”列，它给出了模型对每个可能的输出的准确率的估计。可以看出在这些情况下，有几个样本的值可以肯定是 3 而不是 1。

敏锐的读者可能已经注意到了，尽管只有 7 种可能的结果，而概率向量实际上有 8 个值。向量中索引 1~7 的值分别表示结果为 1~7 的概率。然而，索引 0 也有一个值，它总是显示概率为“0.0”。我们可以忽略它，因为 0 根本就不是一个有效的结果。把这种信息表示为向量就有这样的小毛病，值得我们注意一下。

基于这段代码的观察，该模型还欠火候，预测结果常常出错。与 ALS 的实现一样，决策树分类器的实现有几个超参数需要调整，这段代码中使用的都是默认值。这里测试集可用于对默认超参数训练出的模型的期望准确率做无偏估计。

`MulticlassClassificationEvaluator` 能计算准确率和其他评估模型预测质量的指标。Spark MLlib 中评估器的作用就是以某种方式评估输出 DataFrame 的质量，`MulticlassClassificationEvaluator` 就是评估器的一个例子。

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val evaluator = new MulticlassClassificationEvaluator().
  setLabelCol("Cover_Type").
  setPredictionCol("prediction")

evaluator.setMetricName("accuracy").evaluate(predictions)
evaluator.setMetricName("f1").evaluate(predictions)

...
0.6976371385502989
0.6815943874214012
```

在给出“label”列（目标，或已知的正确输出值）以及预测结果列的列名后，评估器发现这两列有 70% 的匹配率，这就是这个分类器的准确率了。它可以计算其他相关的度量，比如 F1 值（https://en.wikipedia.org/wiki/F1_score）。准确率在这里被用于评价分类器。

单个的准确率可以很好地概括分类器输出的好坏，然而有时候**混淆矩阵**（confusion matrix）会更有效。混淆矩阵是一个 $N \times N$ 的表，N 代表可能的目标值的个数。因为我们的目标值有 7 个分类，所以是一个 7×7 的矩阵，每一行代表数据的真实归属类别，每一列按顺序依次代表预测值。第 i 行和第 j 列的条目表示数据中真正归属第 i 个类别却被预测为第 j 个类别的数据总量。因此，正确的预测是沿着对角线计算的，而非对角线元素代表错误预测。

幸运的是，Spark 提供了用于计算混淆矩阵的代码；不幸的是，这个代码是基于操作 RDD 的旧版 MLlib API 实现的。不过，这并不是什么大问题，DataFrame 和 Dataset 可以很容易地转换成 RDD 来使用这些旧版 API。这边的代码中，用 `MulticlassMetrics` 去处理包含预测结果的 DataFrame 是很合适的。

```
import org.apache.spark.mllib.evaluation.MulticlassMetrics

val predictionRDD = predictions.
  select("prediction", "Cover_Type").
  as[(Double,Double)]. ❶
  rdd ❷
val multiclassMetrics = new MulticlassMetrics(predictionRDD)
multiclassMetrics.confusionMatrix

...
143125.0  41769.0   164.0    0.0    0.0   0.0  5396.0
65865.0   184360.0  3930.0   102.0  39.0  0.0  677.0
0.0       5680.0    25772.0  674.0  0.0   0.0  0.0
0.0       21.0      1481.0   973.0  0.0   0.0  0.0
87.0      7761.0    648.0    0.0    69.0  0.0  0.0
0.0       6175.0    8902.0   559.0  0.0   0.0  0.0
8058.0    24.0      50.0     0.0    0.0   0.0  10395.0
```

❶ 转换成 Dataset。

❷ 转换成 RDD。

你得到的值可能稍有不同。构造决策树过程中的一些随机选项会导致分类结果稍有不同。

对角线上的次数多是好的。但也确实出现了一些分类错误的情况，比如分类器甚至没有将任何一个样本类别预测为 5。

当然，计算混淆矩阵之类，也可以直接使用 DataFrame API 中一些通用的操作，而不再需要依赖专门的方法。

```
val confusionMatrix = predictions.
  groupBy("Cover_Type").
  pivot("prediction", (1 to 7)).
  count().
  na.fill(0.0). ❶
  orderBy("Cover_Type")

confusionMatrix.show()

...
+----------+------+------+-----+---+---+---+-----+
|Cover_Type|     1|     2|    3|  4|  5|  6|    7|
+----------+------+------+-----+---+---+---+-----+
|       1.0|143125| 41769|  164|  0|  0|  0| 5396|
|       2.0| 65865|184360| 3930|102| 39|  0|  677|
|       3.0|     0|  5680|25772|674|  0|  0|    0|
|       4.0|     0|    21| 1481|973|  0|  0|    0|
```

```
|       5.0|    87|  7761|  648|  0| 69|  0|    0|
|       6.0|     0|  6175| 8902|559|  0|  0|    0|
|       7.0|  8058|    24|   50|  0|  0|  0|10395|
+----------+------+------+-----+---+---+---+-----+
```

❶ 用 `0` 替换 `null`。

微软 Excel 的用户可能已经发觉，计算混淆矩阵与计算透视表十分相似。透视表（https://en.wikipedia.org/wiki/Pivot_table）对两个维度上的值进行分组，这些值会变成输出的行和列，然后对这些分组进行计数之类的聚合计算。许多数据库都有 PIVOT 这个函数，Spark SQL 也支持 PIVOT 函数。这种方式更优雅、更强大。

虽然 70% 的准确率听起来还不错，但我们还不能立马看出这个准确率是优秀还是糟糕。作为基准，一个朴素方法的准确率是多少呢？即使是一个坏了的时钟，每天也会有两次显示的时间是正确的。类似地，为每个样本随便猜一个类别，偶尔也能得到正确答案。

按照类别在训练集中出现的比例来预测类别，我们来构建一个“分类器”。举例来说，如果类型 1 在训练集中占 30%，那么分类器就有 33% 的概率猜测类型为“1”。每次分类的准确率将和一个类型在交叉验证集中出现的次数成正比。如果测试集的 40% 是植被类型 1，那么全猜“1”就有 40% 的准确率。植被类型 1 有 30%×40%=12% 的概率被猜对，并为整体准确率贡献了 12%。将所有类别在训练集和交叉验证集出现的概率相乘，然后把结果相加，我们就得到了一个对准确率的评估：

```
import org.apache.spark.sql.DataFrame

def classProbabilities(data: DataFrame): Array[Double] = {
  val total = data.count()
  data.groupBy("Cover_Type").count(). ❶
    orderBy("Cover_Type"). ❷
    select("count").as[Double]. ❸
    map(_ / total).
    collect()
}

val trainPriorProbabilities = classProbabilities(trainData)
val testPriorProbabilities = classProbabilities(testData)
trainPriorProbabilities.zip(testPriorProbabilities).map { ❹
  case (trainProb, cvProb) => trainProb * cvProb
}.sum

...
0.3771270477245849
```

❶ 计算数据中每个类别的样本数。

❷ 对类别的样本数进行排序。

❸ 转换成 Dataset。

❹ 对训练集和测试集中键值对的乘积求和。

随机猜测的准确率为 37%，所以我们前面得到 70% 的准确率看起来还不错。但是这个 70% 的准确率是用默认超参数取得的。如果在决策树构建过程中试试超参数的其他值，准确率还可以提高。

4.8 决策树的超参数

第 3 章中，ALS 算法提供了几个超参数。我们先构造超参数取不同值时的不同组合的模型，然后用某个指标评估每个组合结果的质量，通过这种方式来选择超参数值。这里我们采用相同的过程，但指标由 AUC 改为多元分类准确率。这里控制决策树选择过程的超参数也很不一样，分别为最大深度、最大桶数、不纯性度量和最小信息增益。

最大深度只是对决策树的层数做出限制，它是分类器为了对样本进行分类所做的一连串判断的最大次数。限制判断次数有利于避免对训练数据产生过拟合，这一点我们在前面宠物店的例子中已有说明。

决策树算法负责为每层生成可能的决策规则，比如在宠物店示例中，这些决策规则类似“重量 ⩾ 100”或者“重量 ⩾ 500”。决策总是采用相同形式：对数值型特征，决策采用特征 ⩾值的形式；对类别型特征，决策采用特征在（值 1，值 2，…）中的形式。因此，要尝试的决策规则集合实际上是可以嵌入决策规则中的一系列值。Spark MLlib 的实现把决策规则集合称为“桶”（bin）。桶的数目越多，需要的处理时间越多，但找到的决策规则可能更优。

什么因素会促使产生好的决策规则？直观上讲，好的决策规则应该通过目标类别值对样本做出有意义的划分。比如，如果一个规则将 Covtype 数据集划分为两个子集，其中一个子集的样本全部属于类别 1~3，第二个子集中的样本则都属于和类别 4~7，那么它就是一个好规则，因为这个规则清楚地把一些类别和其他类别分开。如果样本集用一个决策规则划分，划分前后每个集合各类型的不纯性程度没有改善，那么这个规则就没什么价值。沿着该决策规则的分支走下去，每个目标类别的可能取值的分布仍然是一样的，因此实际上它在构造可靠的分类器方面没有任何进步。

换句话说，好规则把训练集数据的目标值分为相对是同类或“纯”（pure）的子集。选择最好的规则也就意味着最小化规则对应的两个子集的不纯性（impurity）。不纯性有两种常用的度量方式：Gini 不纯度（https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity）或熵（https://en.wikipedia.org/wiki/Entropy_(information_theory)）。

Gini 不纯度直接和随机猜测分类器的准确率相关。在每个子集中，它就是对一个随机挑选的样本进行随机分类时分类错误的概率（随机挑选样本和随机分类时要参照子数据集的类别分布）。这就是用 1 减去每个类别的比例与自身的乘积之和。假设子数据集包含 N 个类

别的样本，p_i 是类别 i 的样本所占比例，于是可以得到如下 Gini 不纯度公式：

$$I_G(p) = 1 - \sum_{i=1}^{N} p_i^2$$

如果子数据集中所有样本都属于一个类别，则 Gini 不纯度的值为 0，因为这个子数据集完全是“纯”的。当子数据集中的样本来自 N 个不同的类别时，Gini 不纯度的值大于 0，并且在每个类别的样本数都相同时达到最大，也就是最不纯的情况。

熵是另一种度量不纯性的方式，它来源于信息论。解释熵的本质更困难，但熵代表了子集中目标取值集合对子集中的数据所做的预测的不确定程度。如果子集只包含一个类别，则是完全确定的，熵为 0。相反，如果一个子集包含了所有可能的类别，那么对该子集进行预测有很大的不确定性，因为数据的目标值是各种各样的。这就意味着这个子集有较大的熵。因此，较小的熵，就像较小的 Gini 不纯度一样，是比较好的。熵可以用以下熵计算公式定义：

$$I_E(p) = \sum_{i=1}^{N} p_i \log\left(\frac{1}{p}\right) = -\sum_{i=1}^{N} p_i \log(p_i)$$

> 有意思的是，不确定性是有单位的。由于取自然对数（以 e 为底），熵的单位是**纳特**（nat）。相对于以 e 为底的纳特，我们更熟悉它对应的**比特**（以 2 为底取对数即可得到）。它实际上度量的是信息，因此在使用熵的决策树中，我们也常说决策规则的**信息增益**。

不同的数据集上对于挑选好的决策规则方面，这两个度量指标各有千秋。Spark 的实现默认采用 Gini 不纯度。

最后，最小信息增益是指一种超参数，它会导致最小信息增益，或最小不纯度降低。在改善子集合的不纯性方面不达标的决策规则将不被采用。与通过减少最大深度一样，这也有利于避免过拟合，因为对训练集没有什么区分度的决策规则，实际上对区分将来的数据也没什么帮助。

4.9 决策树调优

采用哪个不纯性度量所得到的决策树的准确率更高，或者最大深度或桶数取多少合适，从数据上看，回答这些问题是困难的。幸运的是，我们可以让 Spark 来尝试这些值的许多组合并报告结果，就像第 3 章所做的那样。

首先，有必要构建一个管道，用于封装与上面相同的两个步骤。创建 `VectorAssembler` 和 `DecisionTreeClassifier`，然后将这两个 `Transformer` 串起来，我们就可以得到一个单独的 `Pipeline` 对象，这个 `Pipeline` 对象可以将前面的两个操作表示成一个：

```
import org.apache.spark.ml.Pipeline

val inputCols = trainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()).
  setLabelCol("Cover_Type").
  setFeaturesCol("featureVector").
  setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, classifier))
```

当然，管道可以更长更复杂，但用起来还是一样方便。现在，我们还可以定义使用 Spark ML API 内建支持的 `ParamGridBuilder` 来测试超参数的组合。现在需要定义评价指标——什么样的超参数才是“最好的”。这里再一次用到 `MulticlassClassificationEvaluator`。

```
import org.apache.spark.ml.tuning.ParamGridBuilder

val paramGrid = new ParamGridBuilder().
  addGrid(classifier.impurity, Seq("gini", "entropy")).
  addGrid(classifier.maxDepth, Seq(1, 20)).
  addGrid(classifier.maxBins, Seq(40, 300)).
  addGrid(classifier.minInfoGain, Seq(0.0, 0.05)).
  build()

val multiclassEval = new MulticlassClassificationEvaluator().
  setLabelCol("Cover_Type").
  setPredictionCol("prediction").
  setMetricName("accuracy")
```

这意味着对 4 个超参数来说，每个超参数的两个值都要构建和评估一个模型，共计 16 种超参数组合，会训练出 16 个模型，并使用多分类准确率对这些模型进行评估。最后，`TrainValidationSplit` 将这些组件拼在一起，形成一个管道。这个管道可以构建模型、模型评估指标并尝试不同的超参数，然后在训练数据上使用模型评价指标对每个模型进行评估。值得注意的是，这里也可以用 `CrossValidator` 执行完整的 k 路交叉验证，但是要额外付出 k 倍的代价，并且在大数据的情况下意义不大。所以在这里 `TrainValidationSplit` 就够用了：

```
import org.apache.spark.ml.tuning.TrainValidationSplit

val validator = new TrainValidationSplit().
  setSeed(Random.nextLong()).
  setEstimator(pipeline).
  setEvaluator(multiclassEval).
  setEstimatorParamMaps(paramGrid).
  setTrainRatio(0.9)

val validatorModel = validator.fit(trainData)
```

依硬件情况而定，这段代码将花费几分钟或更久的时间，因为需要构建并评估许多模型。需要注意的是，参数 `trainRatio` 被设置为 `0.9`，这意味着训练数据实际上被 `TrainValidationSplit` 划分成 90% 与 10% 的两个子集。前面 90% 的数据将用于训练每个模型，剩下的 10% 的数据将作为交叉验证集对模型进行评估。如果它已经拿出了一些数据用于评估模型，那么我们为什么还要取出 10% 的原始数据构建一个测试集呢？

如果说交叉验证集的目的是评估适合**训练**集的**参数**，那么测试集的目的是评估适合交叉验证集的**超参数**。也就是说，测试集保证了对最终选定的超参数及模型准确率的无偏估计。

假如这个程序得到的最优模型能在交叉验证集上达到 90% 的准确率，那么期望它在未来的数据上达到 90% 准确率是合理的。但是，模型在构建过程中还有一个随机元素。最好的模型和评估结果可能还需要一点儿运气的成分，因此准确率评估可能有一些乐观。换句话说，超参数也可能有过拟合现象。

要想真正评估这个最佳模型在将来的样本上的表现，当然需要在没有用于训练的样本上进行评估。但是，我们也需要避免使用在评估环节中用过的交叉验证集样本。这也就是需要把第三个子集即测试集留在一边的原因。

`validator` 的结果包含它找到的最优模型。该模型本身是 `validator` 所能找到的总体最优管道的一种表示，这一点可以从我们提供的一个管道实例看出。为了知道 `DecisionTreeClassifier` 选择的参数，我们需要手动从结果 `PipelineModel` 中提取 `DecisionTreeClassificationModel` 的实例，这是管道的最后一步了。

```
import org.apache.spark.ml.PipelineModel

val bestModel = validatorModel.bestModel
bestModel.asInstanceOf[PipelineModel].stages.last.extractParamMap

...
{
	dtc_9136220619b4-cacheNodeIds: false,
	dtc_9136220619b4-checkpointInterval: 10,
	dtc_9136220619b4-featuresCol: featureVector,
	dtc_9136220619b4-impurity: entropy,
	dtc_9136220619b4-labelCol: Cover_Type,
	dtc_9136220619b4-maxBins: 40,
	dtc_9136220619b4-maxDepth: 20,
	dtc_9136220619b4-maxMemoryInMB: 256,
	dtc_9136220619b4-minInfoGain: 0.0,
	dtc_9136220619b4-minInstancesPerNode: 1,
	dtc_9136220619b4-predictionCol: prediction,
	dtc_9136220619b4-probabilityCol: probability,
	dtc_9136220619b4-rawPredictionCol: rawPrediction,
	dtc_9136220619b4-seed: 159147643
}
```

这包含了很多拟合模型的信息，但它也告诉我们，“熵”作为不纯度的度量是最有效的，而

最大深度 20 比 1 好，也在我们意料之中。最好的模型仅拟合到 40 个桶（bin），这一点倒可能让人有些意外，但这也可能意味着 40 个桶已经“足够好了”，而不是说拟合到 40 个桶比 300 个桶“更好”。最后，`minInfoGain` 的值为 `0`，这比不为零的最小值要更好，因为这可能意味着模型更容易欠拟合（underfit），而不是过拟合（overfit）。

你可能想知道，是否有可能得到每一种超参数组合训练出来的模型的准确率。超参数和评估结果分别用 `getEstimatorParamMaps` 和 `validationMetrics` 获得。将二者组合起来，我们就可以按指标排序并显示所有参数组合：

```
val validatorModel = validator.fit(trainData)

val paramsAndMetrics = validatorModel.validationMetrics.
  zip(validatorModel.getEstimatorParamMaps).sortBy(-_._1)

paramsAndMetrics.foreach { case (metric, params) =>
    println(metric)
    println(params)
    println()
}

...
0.9138483377774368
{
        dtc_3e3b8bb692d1-impurity: entropy,
        dtc_3e3b8bb692d1-maxBins: 40,
        dtc_3e3b8bb692d1-maxDepth: 20,
        dtc_3e3b8bb692d1-minInfoGain: 0.0
}

0.9122369506416774
{
        dtc_3e3b8bb692d1-impurity: entropy,
        dtc_3e3b8bb692d1-maxBins: 300,
        dtc_3e3b8bb692d1-maxDepth: 20,
        dtc_3e3b8bb692d1-minInfoGain: 0.0
}
...
```

这个模型在交叉验证集中达到的准确率是多少？最后，在测试集中能达到什么样的准确率？

```
validatorModel.validationMetrics.max
multiclassEval.evaluate(bestModel.transform(testData)) ❶

...
0.9138483377774368
0.9139978718291971
```

❶ `bestModel` 是一个完整的管道。

两个结果都是 91%。交叉验证集上的评估结果常常能提供不错的起点。事实上，测试集通

常会有与交叉验证集相似的表现。

现在该重新回顾一下过拟合的问题了。如前所述，我们可能构造这样一棵决策树：它的深度非常深，非常复杂，它能很好地甚至是完美拟合给定的训练样本，但不能把这种准确率推广到其他样本上，因为它过于紧密地拟合了训练样本中的细微特质和噪声。过拟合问题不只是在决策树算法中存在，而是大多数机器学习算法普遍存在的问题。

当决策树有过拟合问题时，在与模型拟合相同的训练数据上它的准确率很高，但在其他样本上准确率很低。在本章示例中，最终模型在其他新样本上的准确率大约为 91%。当然，通过 `trainData`，我们就能轻松地在训练数据上评估准确率。这时的准确率大约为 95%。

差别不是很大，但也显示决策树在一定程度上存在对训练数据的过拟合。减小最大深度可能会使过拟合问题有所改善。

4.10 重谈类别型特征

目前为止，上述代码示例将所有输入特征隐式地作为数值型处理（尽管“Cover_Type”被编码为数字，实际上被正确地视为类别型特征）。这并不是完全错误的，因为这里的类别型特征已经用 one-hot 方式编码成了多个二元的 0/1 值。把这些单个的特征当作数值型来处理并没有什么问题，因为任何基于数值型特征的决策规则都需要选择 0 或 1 作为其阈值，并且因为所有的阈值都是 0 或 1，所以都是等价的。

当然，这种编码迫使决策树算法在底层要单独考虑类别型特征的每一个值。因为像土壤类型这样的特征被分解成了许多特征，而且决策树单独处理这些特征，所以就更难将这些相关的土壤类型信息关联起来。

例如，9 种不同的土壤类型实际上是 Leighcan 族中的一部分，而决策树应该利用它们之间的这种关联。如果土壤类型被编码为一个具有 40 个值的单一类别型特征，那么树就可以用于直接描述“土壤类型是否是 9 个 Leighton 族类型之一”。然而，当被编码为 40 个特征时，要取得相同效果，树就必须学习一个决策序列，其中包括 9 个关于土壤类型的决策，这种表现形式可能会得到更好、更高效的决策。

然而，用 40 个数值型特征表示一个带 40 个值的类别型特征，将导致内存使用用量增加，同时处理速度也会变慢。

如果取消数据集已经完成的 one-hot 编码，情况会怎样？例如，4 列编码的荒原类型可以替换为 1 列用数字 0~3 编码的方式，像“Cover_Type”列一样。

```
import org.apache.spark.sql.functions._

def unencodeOneHot(data: DataFrame): DataFrame = {
```

```
  val wildernessCols = (0 until 4).map(i => s"Wilderness_Area_$i").toArray

  val wildernessAssembler = new VectorAssembler().
    setInputCols(wildernessCols).
    setOutputCol("wilderness")

  val unhotUDF = udf((vec: Vector) => vec.toArray.indexOf(1.0).toDouble) ❶

  val withWilderness = wildernessAssembler.transform(data).
    drop(wildernessCols:_*). ❷
    withColumn("wilderness", unhotUDF($"wilderness")) ❸

  val soilCols = (0 until 40).map(i => s"Soil_Type_$i").toArray

  val soilAssembler = new VectorAssembler().
    setInputCols(soilCols).
    setOutputCol("soil")

  soilAssembler.transform(withWilderness).
    drop(soilCols:_*).
    withColumn("soil", unhotUDF($"soil"))
}
```

❶ 注意用户定义函数的定义。

❷ 删除不再需要的 one-hot 列。

❸ 用数字编码的列替换其同名列。

这里我们用 `VectorAssembler` 将 4 个荒野类型列与 40 个土壤类型列组合成两个向量列。这些向量除了一个位置上的值是 1，其余的都是 0。DataFrame 没有提供直接完成这个任务的函数，因此我们必须定义自己的 UDF 来操作这些列。这个 UDF 将这两个新列变成了我们所需类型的值。

从这里开始，我们都可以使用与上面几乎相同的过程，来调整构建在该数据上的决策树模型的超参数，并选择和评估最优的模型。但是有一个重要的不同点，两个新的数值列没有任何信息表明它们实际上是类别型特征值的编码。把它们当作数值型是错误的，因为它们的大小是没有意义的。然而，这种处理方式似乎还能奏效，好像什么事情都没发生一样，不过这些特征的信息可能在处理过程中就完全丢失了。

Spark MLlib 内部可以存储每列额外的元数据信息。这些数据的细节通常对调用者来说是隐藏的，诸如列是否对类别型特征值编码，以及类别型编码有多少不同的值。为了添加这些元数据，有必要通过 `VectorIndexer` 存入这些数据。`VectorIndexer` 的目标是将输入变成有正确标签的类别型特征列。尽管我们已经做了很多工作来将类别型特征转换为从 0 开始的索引值，`VectorIndexer` 将会负责生成元数据。

需要将以下步骤加入 `Pipeline`：

```
import org.apache.spark.ml.feature.VectorIndexer

val inputCols = unencTrainData.columns.filter(_ != "Cover_Type")
val assembler = new VectorAssembler().
  setInputCols(inputCols).
  setOutputCol("featureVector")

val indexer = new VectorIndexer().
  setMaxCategories(40). ❶
  setInputCol("featureVector").
  setOutputCol("indexedVector")

val classifier = new DecisionTreeClassifier().
  setSeed(Random.nextLong()).
  setLabelCol("Cover_Type").
  setFeaturesCol("indexedVector").
  setPredictionCol("prediction")

val pipeline = new Pipeline().setStages(Array(assembler, indexer, classifier))
```

❶ ≥ 40，因为土壤类型有 40 个值。

该方法假定训练集中类别型特征的所有可能值出现了至少一次。也就是说，只有当所有 4 个土壤值和所有 40 个荒原类型值都出现在训练集中时，让所有可能值都有映射，这个方法才能正常工作。在本数据集中，这些假设正好都满足，但是在小的训练数据集中可能就不是这样了，某一类标签可能很少出现。在这些情况下，可能需要手动创建并添加一个 `VectorIndexerModel`，并手动提供完整的值映射。

除此以外，过程与之前是一样的。你会发现它选择了一个类似的最优模型，但是在测试集上的准确率却有 93%。将类别型特征按其实际类型处理后，分类器的准确率提高了近 2%。

4.11　随机决策森林

如果你一步步运行本章示例代码，可能已经注意到运行结果和本书代码中给出的结果稍有不同。这是在决策树构建过程中由随机因素造成的，在决定采用什么数据和尝试哪些决策规则时都有这些随机因素的影响。

在决策树的每层，算法并不会考虑所有可能的决策规则。如果在每层上都要考虑所有可能的决策规则，算法的运行时间将无法想象。对一个有 N 个取值的类别型特征，总共有 2^N-2 个可能的决策规则（除空集和全集以外的所有子集）。即使对于一个一般大的 N，这也将创建数十亿候选决策规则。

相反，决策树使用一些启发式策略，来决定哪些是需要实际考虑的少数规则。在选择规则的过程中也涉及一些随机性；每次只考虑随机选择少数特征，而且只考虑训练数据中一个随机子集。在牺牲一些准确率的同时换回了速度的大幅提升，但也意味着每次决策树算法

构造的树都不相同。这是件好事。

这也能解释为何集体的智慧常常比个体预测要更准确。为了说明问题，我们来做个快速测验：伦敦运营的黑色出租车数量有多少？

请猜一下，先不要偷看答案。

正确答案是约 19 000，我猜 10 000，这离正确答案差了很多。因为我猜的数比较小，所以你猜的可能比我大，这样我们平均一下就可能更准确了。这里貌似又有点儿“趋均值回归”的味道了。我随便问了办公室里的 13 个人，大家的平均值是 11 170，这个答案确实要更接近正确答案。

要取得这种效应，关键是每个人猜的时候要独立，互不影响。（你没偷看答案，是吧？）如果大家事先都已经统一过意见并用同一个方法猜，这个练习就没有意义了，因为大家猜的答案都一样，而且这个相同的答案可能错得离谱。如果在你猜之前，我把我的情况告诉你，这会影响你的猜测，那么咱俩的平均数可能并不会更准确，甚至会更糟。

基于上述考虑，树应该不止有一棵，而是有很多棵，每一棵都能对正确目标值给出合理、独立且互不相同的估计。这些树的集体平均预测应该比任一个体预测更接近正确答案。正是由于决策树构建过程中的**随机性**，才有了这种独立性，这就是**随机决策森林**的关键所在。

构建多棵决策树的方式注入了随机因素，每棵树使用了数据的一个不同的随机子集，甚至使用随机的特征子集。这样得到的森林在整体上就不那么容易产生过拟合。如果某个特征包含噪声数据，或只针对**训练集**有预测性，则这种预测性是有误导性质的。采用随机森林后大多数树在大多数时候将因此不会考虑这个问题特征。大多数的树将不会拟合噪声，因此它们的“票数”将超过那些拟合噪声的树。

随机决策森林的预测只是所有决策树预测的加权平均。对于类别型目标，这就是得票最多的类别，或有决策树概率平均后的最大可能值。随机决策森林和决策树一样也支持回归问题，这时森林做出的预测就是每棵树预测值的平均。

虽然随机决策森林是一个更加强大且更加复杂的分类技术，但好在使用管道的方式和本章之前论述的并没有什么差异。只需要使用 `RandomForestClassifier` 替代 `DecisionTreeClassifier`，后面采用同样的处理方式。想要使用随机森林，之前的代码和 API 就够用了。

```
import org.apache.spark.ml.classification.RandomForestClassifier

val classifier = new RandomForestClassifier().
  setSeed(Random.nextLong()).
  setLabelCol("Cover_Type").
  setFeaturesCol("indexedVector").
  setPredictionCol("prediction")
```

请注意，这个分类器有另外一个超参数：要构建的决策树的个数。与超参数 maxBins 一样，在某个临界点之前，该值越大应该就能获得越好的效果。然而，代价是构造多棵决策树的时间比建造一棵的时间要长很多倍。

从类似的调优过程得到的最优随机决策森林模型，其准确率在之前已经提高约 2% 的基础上，又提高到 95%。也就是说，与之前构建的最优的决策树相比，错误率降低了 28%——从 7% 降到了 5%。如果对模型进一步调优，结果可能还会更好。

顺便说一句，此时我们对特征的重要性的理解更准确了：

```
import org.apache.spark.ml.classification.RandomForestClassificationModel

val forestModel = bestModel.asInstanceOf[PipelineModel].
  stages.last.asInstanceOf[RandomForestClassificationModel]

forestModel.featureImportances.toArray.zip(inputCols).
  sorted.reverse.foreach(println)

...
(0.28877055118903183,Elevation)
(0.17288279582959612,soil)
(0.12105056811661499,Horizontal_Distance_To_Roadways)
(0.1121550648692802,Horizontal_Distance_To_Fire_Points)
(0.08805270405239551,wilderness)
(0.04467393191338021,Vertical_Distance_To_Hydrology)
(0.04293099150373547,Horizontal_Distance_To_Hydrology)
(0.03149644050848614,Hillshade_Noon)
(0.028408483578137605,Hillshade_9am)
(0.027185325937200706,Aspect)
(0.027075578474331806,Hillshade_3pm)
(0.015317564027809389,Slope)
```

在大数据的背景下，随机决策森林非常有吸引力，因为决策树往往是独立构造的，诸如 Spark 和 MapReduce 这样的大数据技术本质上适合数据并行问题。也就是说，总体答案的每个部分可以通过在部分数据上独立计算来完成。随机决策森林中决策树可以并且应该只在特征子集或输入数据子集上进行训练，基于这个事实，决策树构造的并行化就很简单了。

4.12　进行预测

构建分类器虽然有趣且简单，但它不是最终目的。我们的目的是利用它进行预测。我们之前的辛苦努力在这里将得到回报，而且做起来相对非常容易。

由此得到的“最优模型”实际上是包含所有操作的整个管道，其中包括如何对输入进行转换以适于模型处理，以及用于预测的模型本身。它可以接受新的 DataFrame 作为输入。它与我们刚开始时使用的 DataFrame 数据的唯一区别就是缺少“Cover_Type”列。波尔先生说，当进行预测时，特别是预测未来时，输出当然是未知的。

为了证明这一点，可以尝试删除测试集输入的“Cover_Type”列，并进行一次预测：

```
bestModel.transform(unencTestData.drop("Cover_Type")).select("prediction").show()

...
+----------+
|prediction|
+----------+
|       6.0|
+----------+
```

结果应该为 6.0，对应原始 Covtype 数据集中的类别 7（原始特征从 1 开始）。显然，算法预测示例中讨论的地块植被类型为“高山矮曲林”（Krummholz）。

4.13 小结

本章介绍了相互关联的两类重要的机器学习算法：分类和回归。我们还一同介绍了模型构造和调优的一些基本概念：特征、向量、训练和交叉验证。使用 Covtype 数据集和 Spark MLlib 中实现的决策树和决策森林算法，本章演示了如何根据位置和土壤类型等信息预测森林植被的类型。

和第 3 章一样，我们还可以进一步研究超参数对模型准确率的影响。在选择决策树超参数时，我们往往会为了更高的准确率而宁愿花费更长时间：增加桶和树的数目通常能得到更高的精度，但精度的提高在到达一定程度之后提高的幅度会越来越小。

本章构建的分类器结果非常准确。一般情况下，准确率超过 95% 是很难达到的。通常，通过包括更多特征，或将已有特征转换成预测性更好的形式，我们可以进一步提高准确率。在分类器模型的迭代式改进过程中，我们常常这样反复尝试。比如，对本章的数据集，距离地表水的水平和垂直距离这两个特征可以生成第三个特征：离地表水的直线距离。或者，如果能收集到更多数据，为了提高准确率，我们可能会尝试增加更多特征，比如土壤湿度。

当然，用 Covtype 数据集预测森林植被类型只是预测问题的一种类型，现实中我们还有其他的预测问题。比如，有些问题要求预测连续型的数值，而不是类别型值。对于这类回归问题，本章介绍的许多分析方法和代码照样适用，不过要使用 `RandomForestRegressor` 类。

再者，分类和回归算法不只包括决策树和决策森林，Spark MLlib 实现的算法也不限于决策树和决策森林。对分类问题，Spark MLlib 提供的实现包括：

- 朴素贝叶斯（https://en.wikipedia.org/wiki/Naive_Bayes_classifier）
- Gradient boosting（https://en.wikipedia.org/wiki/Gradient_boosting）
- logistic 回归（https://en.wikipedia.org/wiki/Logistic_regression）
- 多层感知机（Multilayer perceptron，https://en.wikipedia.org/wiki/Multilayer_perceptron）

是的，你没看错，logistic 回归是一种分类技术。logistic 回归底层通过预测类别的连续型概率函数来进行分类。细节内容我们不必理解。

这些算法与决策树和决策森林区别很大。但是，其中许多元素还是一样的：它们是管道的一部分，操作 DataFrame 中的列，并且需要将输入数据划分为训练集、交叉验证集和测试集来选择超参数。对这些其他算法，我们可以用相同的通用原理为分类和决策问题建模。

这些都是监督学习的例子。如果某些目标值或全部目标值都是未知的，情况又会怎样？下一章我们将探寻解决之道。

第 5 章

基于K均值聚类的网络流量异常检测

作者：肖恩·欧文

> 据我们所知，有“已知的已知”；有些事，我们知道我们知道。我们也知道，有“已知的未知”，也就是说，有些事，我们现在知道我们不知道。但是，同样存在“不知的不知”；有些事，我们不知道我们不知道。
>
> ——Donald Rumsfeld

分类和回归技术很强大，在机器学习领域被广泛研究。第 4 章我们讲述了如何用分类器预测未知值。这里有一点很关键：为了预测新数据的未知值，必须事先给定许多样本并且样本的目标值是已知的。分类器仅在数据科学家知道他们要寻找什么，并且可以提供大量包含输入及正确输出的样本数据的情况下才能发挥作用。由于在学习过程中，对每个输入样本都给出正确的输出值作为指导，所以分类和回归都属于**监督学习技术**（https://en.wikipedia.org/wiki/Supervised_learning）。

然而，有时样本仅能给出部分正确输出，甚至无法给出输出。这些情况下，分类和回归技术将无法使用。考虑到电子商务网站的情况，我们要按照购物习惯和偏好将顾客分组。这里输入特征是顾客的购买记录、点击记录和个人信息等，输出则是顾客所属的群体，其中可能有一群顾客时尚意识较强，还可能有一群更追求性价比。

现在要求你确定每个新顾客所属的目标群体。如果采用分类等监督学习技术，你可能很快就会发现缺乏先验知识的问题：“哪些顾客时尚意识较强”这个信息是没有的。“时尚意识

较强”这个顾客群组对电子商务网站是否有意义，你甚至都不确定。

这时我们就要用到**非监督学习技术**（https://en.wikipedia.org/wiki/Unsupervised_learning）了！因为目标值是未知的，所以非监督学习技术不会学习如何预测目标值。但是，它可以学习数据的结构并找出相似输入的群组，或者学习哪些输入类型可能出现，哪些类型不可能出现。本章将通过 MLlib 实现的聚类算法来介绍非监督学习技术。

5.1 异常检测

顾名思义，异常检测就是要找出不寻常的情况。如果已经知道“异常”代表什么涵义，我们就能通过监督学习轻松地检测出数据集中的异常。在样本的输入被标记为“正常”和“异常”后，算法就能通过学习样本来区分“正常”和“异常”了。然而本质上异常属于“未知的未知”，也就是说，在我们观察并理解了异常以后，它就不再是异常了。

异常检测常用于检测欺诈、网络攻击、服务器及传感设备故障。在这些应用中，我们要能够找出以前从未见过的新型异常，如新欺诈方式、新入侵方法或新服务器故障模式。

这些应用要用到非监督学习技术，通过学习，它们知道什么是正常输入，因此能够找出与历史数据有差异的新数据。这些新数据不一定是攻击或欺诈，它们只是不同寻常，因此值得我们做进一步的调查。

5.2 K均值聚类

聚类是最有名的非监督学习算法，它试图找到数据中的自然群组。一群彼此相似而与其他点不同的数据点往往属于代表某种意义的一个簇群，聚类算法就是要把这些相似的数据划分到同一簇群中。

K 均值聚类（https://en.wikipedia.org/wiki/K-means_clustering）也许是应用最广泛的聚类算法。它试图在数据集中找出 k 个簇群，这里 k 值由数据科学家指定。k 是模型的超参数，其最优值与数据集本身有关。事实上，本章的一个关键点就是如何选择合适的 k 值。

对客户活动或交易数据集来说，“相似”到底代表什么？ K 均值算法有个数据点相距多远的概念。在 K 均值算法中数据点相互距离一般采用欧氏距离。截至本书撰写时，欧氏距离是 Spark MLlib 中支持的唯一距离度量。计算欧氏距离时要求数据点的特征都是数值型。数据点“相似”就是指它们相互间的距离小。

在 K 均值算法中簇群其实就是一个点，即组成该簇的所有点的中心。数据点其实就是由所有数值型特征组成的特征向量，简称向量。因为在欧氏空间中向量被当成了点，所以直接把数据点看成点会更加直观。

簇群的中心称为**质心**（centroid），它是簇群中所有点的算术平均值，因此算法取名 K **均值**。算法开始时选择一些数据点作为簇群的质心。然后把每个数据点分配给最近的质心。接着对每个簇计算该簇所有数据点的平均值，并将其作为该簇的新质心。然后不断重复这个过程。

对 K 均值算法的介绍到此为止，还有值得注意的一些细节内容，我们将在接下来的案例部分再做论述。

5.3　网络入侵

现在，网络攻击越来越多地见诸报端。有些攻击试图通过产生大量网络流量来阻塞计算机上的合法流量。其他一些攻击则试图利用网络软件的缺陷以实现对该计算机的非法访问。虽然识别流量“轰炸”式入侵方式十分简单，但要识别第二种入侵方式则无异于大海捞针。

有些入侵方式的模式是已知的。比如，连续不断地访问某个机器的所有端口，而正常的软件程序是不会这么做的。这往往是攻击的第一步，其目的是要找到计算机上有哪些可能被攻破的服务。

统计对各个端口在短时间内被远程访问的次数，就可以得到一个特征，该特征可以很好地预测端口扫描攻击。如果这种访问次数不多，则属于正常情况；但如果短时间内有好几百个访问，则属于攻击行为。这种方法也适用于其他从网络连接特征中预测网络攻击，这些特征包括发送与接收的字节数和 TCP 错误数等。

但如何处理那些“未知的未知”？最大的威胁可能就来自从未被识别和分类的情况。检测潜在网络入侵就是要识别这些异常，我们并不知道这些连接是不是攻击，但是它们和以往见过的连接都不同。

这里我们可以利用 K 均值之类的非监督学习技术来识别异常网络连接。K 均值可以根据每个网络连接的统计属性进行聚类。从个体上来看结果簇群是没有意义的，但从整体上看，结果簇群定义了历史连接的类型。因此簇群帮助我们界定了正常连接的区域，任何在区域之外的点都是不正常的，因而也就可能属于异常情况。

5.4　KDD Cup 1999数据集

KDD Cup（http://www.kdd.org/kdd-cup）是一项数据挖掘竞赛，每年由美国计算机协会（Association for Computing Machinery，ACM）特别兴趣小组举办。KDD Cup 每年都给出一个机器学习问题和相关数据集，研究人员应邀提交论文，论文将详细说明研究人员各自就该机器学习问题给出的最佳方案。KDD Cup 与之前的 Kaggle 竞赛（https://www.

kaggle.com/）类似。1999 年 KDD Cup 竞赛的主题是网络入侵（http://www.kdd.org/kdd-cup/view/kdd-cup-1999/Tasks），今天我们仍然可以拿到当时的数据集（http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html）。本章将基于 KDD Cup 1999 数据集，利用 Spark 构造一个网络流量异常检测系统。

请切记不要基于 KDD Cup 1999 数据集建立生产系统！该数据集并不一定反映当时网络流量的真实情况，而且即便如此，它反映的网络流量规律也是 17 年前的了。

幸运的是，举办方已经对原始网络流量包进行了加工，数据转换成了每个网络连接的统计信息。数据集大小约为 708 MB，包含 490 万个连接。数据量比较大但也不算特别大，刚好满足本章论述的需要。数据集中每个连接的信息包括发送的字节数、登录次数、TCP 错误数等。数据集为 CSV 格式，每个连接占一行，包含 38 个特征，下面是其中一个连接的样例：

```
0,tcp,http,SF,215,45076,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,
0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,
0.00,0.00,0.00,0.00,0.00,0.00,0.00,normal.
```

以上代表一个 TCP 连接，它访问 HTTP 服务，发送了数据 215 字节，收到数据 45 706 字节，用户登录成功等。其中许多特征代表统计次数，比如第 17 列的 `num_file_creations`。

许多特征取值为 0 或 1，比如第 15 列 `su_attempted`，它们代表某种行为出现与否。这些特征类似第 4 章介绍的用 one-hot 编码的类别型特征，但分类和关联方式有所不同。每个都像是一个“是 / 否”特征，因此可以认为是一个类别型特征。将类别型特征转换为数字并且按大小排序并不适合所有场景。但对于二元类别型特征这种特殊情况，将其映射为 0/1 的数值型特征对于大多数机器学习算法都是没问题的。

其他的特征都代表比率，比如 `dst_host_srv_rerror_rate`，取值的范围为 `[0.0,1.0]`。

注意最后的字段表示类别标号。大多数标号为 `normal.`，但也有一些样本代表各种网络攻击。虽然这些样本可用于学习如何把“已知”的攻击从正常连接中区分开来，但这里要讨论的是异常检测问题，所以我们更关心找出新的潜在攻击，也就是“未知”攻击。因此我们先不在算法中使用这些标号信息。

5.5 初步尝试聚类

将 kddcup.data.gz 数据文件解压并复制到 HDFS 上。像以前一样，这里我们假设文件放在 /user/ds/kddcup.data 目录下。启动 `spark-shell` 并把 CSV 格式数据加载为 DataFrame。这又是一个 CSV 文件，不过这次没有文件头。因此，需要用到附带的文件 kddcup.names 提供的列名。

```
val dataWithoutHeader = spark.read.
  option("inferSchema", true).
  option("header", false).
  csv("hdfs:///user/ds/kddcup.data")

val data = dataWithoutHeader.toDF(
  "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label")
```

先来看看数据集。我们想知道数据有哪些类别标号以及每类样本有多少。以下代码分类统计样本个数，按样本数从多到少排序，然后打印结果：

```
data.select("label").groupBy("label").count().orderBy($"count".desc).show(25)

...
+----------------+-------+
|           label|  count|
+----------------+-------+
|          smurf.|2807886|
|        neptune.|1072017|
|         normal.| 972781|
|          satan.|  15892|
...
|            phf.|      4|
|           perl.|      3|
|            spy.|      2|
+----------------+-------+
```

可以看到数据集中样本有 23 个不同类型，其中 smurf. 和 neptune. 类型的攻击最多。

注意数据中有些特征不是数值型的。比如第二列可能取值 tcp、udp 或 icmp，但是 K 均值聚类算法要求特征为数据型。最后的标号列也不是数值型。我们先简单忽略这些非数值列。

除此以外，对数据进行 K 均值聚类，这与第 4 章中的做法一样。用 VectorAssembler 创建一个特征向量，基于这些特征向量用一个 K 均值实现来创建一个模型，再用一个管道将它们拼接在一起。从得到的模型中，可以提取并检验簇群中心。

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.clustering.{KMeans, KMeansModel}
import org.apache.spark.ml.feature.VectorAssembler

val numericOnly = data.drop("protocol_type", "service", "flag").cache()

val assembler = new VectorAssembler().
  setInputCols(numericOnly.columns.filter(_ != "label")).
  setOutputCol("featureVector")

val kmeans = new KMeans().
  setPredictionCol("cluster").
  setFeaturesCol("featureVector")

val pipeline = new Pipeline().setStages(Array(assembler, kmeans))
val pipelineModel = pipeline.fit(numericOnly)
val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]

kmeansModel.clusterCenters.foreach(println)

...
[48.34019491959669,1834.6215497618625,826.2031900016945,793027561892E-4,...
[10999.0,0.0,1.309937401E9,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...
```

对这些数字做一个直观的解释并不容易，但是每一个数字都表示模型生成的一个簇群中心 [也称为质心（centroid）]。就每个数值输入特征而言，这些值是质心的坐标。

程序输出两个向量，代表 K 均值将数据聚类成 k=2 个簇。对本章的数据集，我们知道连接的类型有 23 个，因此程序肯定没能准确刻画出数据中的不同群组。

这时利用给定的类别标号信息，我们就能直观地看到两个簇中分别包含哪些类型的样本。为此，可以对每个簇中每个标号出现的次数进行计数。

```
val withCluster = pipelineModel.transform(numericOnly)

withCluster.select("cluster", "label").
  groupBy("cluster", "label").count().
  orderBy($"cluster", $"count".desc).
  show(25)

...
+-------+----------------+-------+
|cluster|           label|  count|
+-------+----------------+-------+
|      0|          smurf.|2807886|
|      0|        neptune.|1072017|
|      0|         normal.| 972781|
|      0|          satan.|  15892|
|      0|        ipsweep.|  12481|
...
|      0|            phf.|      4|
|      0|           perl.|      3|
```

```
|      0|           spy.|      2|
|      1|      portsweep.|      1|
+-------+---------------+-------+
```

结果显示聚类根本没有任何作用。簇 1 只有一个数据点！

5.6 *k*的选择

显然将数据集聚类成两个簇是不够的。但究竟聚类成多少个簇合适呢？很明显本章数据集有 23 种不同的入侵模式，因此看起来 *k* 至少应该取 23 或者更大。通常情况，下我们要尝试多个不同的 *k* 值才能找到最好的 *k* 值。但"最好"代表什么涵义呢？

如果每个数据点都紧靠最近的质心，则可认为聚类是较优的。这里的"近"采用欧氏距离（Euclidean distance）定义。这是评估聚类质量的一种简单又常用的方法，使用与所有点之间距离的平均值，有时也可以使用平方距离的平均值。实际上，`KMeansModel` 提供了一个 `computeCost` 方法来计算平方距离的总和，并且很容易用来计算平方距离的平均值。

不幸的是，不像计算多分类指标那样，现在没有一个 `Evaluator` 的实现可以简单地计算这个度量。不过自己写代码来评估几个 *k* 值的聚类成本也不难。请注意，下面的代码需要运行至少 10 分钟。

```
import org.apache.spark.sql.DataFrame

def clusteringScore0(data: DataFrame, k: Int): Double = {
  val assembler = new VectorAssembler().
    setInputCols(data.columns.filter(_ != "label")).
    setOutputCol("featureVector")
  val kmeans = new KMeans().
    setSeed(Random.nextLong()).
    setK(k).
    setPredictionCol("cluster").
    setFeaturesCol("featureVector")
  val pipeline = new Pipeline().setStages(Array(assembler, kmeans))
  val kmeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel]
  kmeansModel.computeCost(assembler.transform(data)) / data.count() ❶
}

(20 to 100 by 20).map(k => (k, clusteringScore0(numericOnly, k))).
  foreach(println)

...
(20,6.649218115128446E7)
(40,2.5031424366033625E7)
(60,1.027261913057096E7)
(80,1.2514131711109027E7)
(100,7235531.565096531)
```

❶ 通过平方距离的总和，计算平方距离的均值（"cost"）。

Scala 通常采用 (`x to y by z`) 这种形式的惯用语法来建立一个数字集合，该集合的元素为闭合区间内的等差数列。这种语法可用于快速建立一系列 *k* 值，如“20, 40, …, 100”，然后对每个值分别执行某项任务。

输出结果显示得分随着 *k* 的增加而降低。注意分数是用科学记数法表示的，第一个值超过了 10^7，不是仅略大于 6。

> 由于聚类结果依赖于随机选择的初始质心，可能你又会看到稍微不同的结果。

但是这个结果没什么稀奇的。随着簇的增加，数据点肯定可以更接近最近的质心。实际上如果 *k* 值等于数据点的个数，由于此时每个点都是自己构成的簇的质心，此时平均距离为 0。

更糟糕的情况是，前面的结果中 *k*=80 时的距离居然比 *k*=60 的距离大。这不应该发生，因为 *k* 取更大值时，聚类的结果应该至少与 *k* 取一个较小值时的结果一样好。问题的原因在于，这种给定 *k* 值的 K 均值算法并不一定能得到最优聚类。K 均值的迭代过程是从一个随机点开始的，因此可能收敛于一个局部最小值，这个局部最小值可能还不错，但并不是全局最优的。

即使采用更加智能的方法来选择初始质心，上述情况依然会存在。“K 均值 ++”和“K 均值 ||”是 K 均值算法的变体，其初始质心算法更容易产生多种多样且相对分散的初始质心，因而更容易得到较好的聚类结果。实际上 Spark MLlib 实现的就是“K 均值 ||”算法（https://stanford.io/1ALCOaN）。但是，不管怎样这里还是有随机选择的因素，所以不能保证全局最优。

k=80 时没能取得最优聚类结果，这可能是随机初始质心所造成的，也可能是由于算法在达到局部最优之前就过早结束了。

增加迭代时间可以优化聚类结果。算法提供了 `setTol()` 来设置一个阈值，该阈值控制聚类过程中簇质心进行有效移动的最小值。降低该阈值能使质心继续移动更长的时间。使用 `setMaxIter()` 增加最大迭代次数也可以防止它过早停止，代价是可能需要更多的计算。

```
def clusteringScore1(data: DataFrame, k: Int): Double = {
  ...
    setMaxIter(40). ❶
    setTol(1.0e-5) ❷
  ...
}
```

```
(20 to 100 by 20).map(k => (k, clusteringScore1(numericOnly, k))).
  foreach(println)
```

❶ 从默认值 20 开始增加。

❷ 默认为 1.0e−4，这里比默认值小。

这时随着 *k* 值的增大，至少结果得分持续下降：

```
(20,1.8041795813813403E8)
(40,6.33056876207124E7)
(60,9474961.544965891)
(80,9388117.93747141)
(100,8783628.926311461)
```

我们要找到 *k* 值的一个临界点，过了这个临界点之后继续增加 *k* 值并不会显著地降低得分，这个点就是 *k* 值 – 得分曲线的拐点。这条曲线通常在拐点之后会继续下行但最终趋于水平。在本示例中，在 *k* 过了 100 这个点之后得分下降还是很明显，所以 *k* 的拐点值应该大于 100。

5.7 基于SparkR的可视化

再次进行聚类之前，我们先停下来，更深入地了解一下数据，这是有好处的。尤其是查看一些数据的散点图是很有帮助的。

Spark 本身没有提供可视化工具，但是流行的开源统计环境 R（https://www.r-project.org/）有我们需要的数据探查和数据可视化工具。此外，Spark 还通过 SparkR（https://spark.apache.org/docs/latest/sparkr.html）提供了一些与 R 的基础集成。这一节我们将简要地演示如何使用 R 和 SparkR 对数据进行聚类，并探查这些产生的簇群。

本书中使用的 SparkR 是 `spark-shell` 的一个变体，用命令 `sparkR` 来运行它。SparkR 运行一个本地的 R 解释器，这一点和 `spark-shell` 本地运行 Scala shell 的变体类似。运行 `sparkR` 的机器需要在本地安装 R，但 Spark 安装包并不包含 R。要安装 R，Ubuntu 这类 Linux 发行版用户可使用 `sudo apt-get install r-base`，macOS 用户可以使用 Homebrew（https://brew.sh/）运行 `brew install R`。

与 R 类似，SparkR 也是一个命令行 shell 环境。要想将其可视化，我们需要在一个可以展示图形的类 IDE 环境中运行这些命令。RStudio（https://www.rstudio.com/）是一种支持 R 的 IDE（也支持 SparkR）；它运行在桌面操作系统上，所以只能在本地机器上的 RStudio 试验 Spark，而不是在集群上。

如果在本地运行 Spark，可以下载并安装 RStudio 的免费版（https://www.rstudio.com/products/rstudio/download2/）。否则即使不在本地运行，本示例中余下的大部分代码也可以通过 `sparkR` 命令行在集群上执行，这样就无法展现可视化效果了。

如果通过 RStudio 执行，我们先启动 IDE。如果本地环境没有设置过 SPARK_HOME 和 JAVA_HOME 的话，需要设置这两个环境变量，让它们分别指向 Spark 和 JDK 的安装目录：

```
Sys.setenv(SPARK_HOME = "/path/to/spark") ❶
Sys.setenv(JAVA_HOME = "/path/to/java")
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "local[*]",
  sparkConfig = list(spark.driver.memory = "4g"))
```

❶ 当然，需要根据实际情况替换路径。

请注意，如果是在命令行上运行 sparkR，上面的步骤就不必要了。对于命令行运行 sparkR 的情况，我们通过命令行配置参数，比如 --driver-memory，这一点与 spark-shell 类似。

SparkR 是本章前面介绍过的 DataFrame 和 MLlib API 的 R 语言包装。因此，可以为数据重新创建一个简单的 K 均值簇群：

```
clusters_data <- read.df("/path/to/kddcup.data", "csv", ❶
                         inferSchema = "true", header = "false")
colnames(clusters_data) <- c( ❷
  "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label")

numeric_only <- cache(drop(clusters_data, ❸
                           c("protocol_type", "service", "flag", "label")))

kmeans_model <- spark.kmeans(numeric_only, ~ ., ❹
                             k = 100, maxIter = 40, initMode = "k-means||")
```

❶ 使用 kddcup.data 的真实路径。

❷ 列名。

❸ 再一次去掉非数值类型的列。

❹ ~ . 表示所有的列。

这里给每个数据点指定簇群是很简单的。上面的操作展示了 SparkR API 的用法，我们可以很自然地将它们和 Spark Core API 中的操作对应起来，不过这些操作使用的是 R 库和类 R

语法。真正的聚类仍旧使用 MLlib 库的实现，它基于 JVM 并使用 Scala。这些操作实际上只是分布式运算的一个句柄或是远程控制器，这些分布式运算并没有在 R 中执行。

R 有自己丰富的数据分析库，也有类似 DataFrame 的概念。因此，有时候为了能够使用这些与 Spark 无关的 R 原生库，把一些数据放入 R 解释器中处理是很有用的。

当然，R 及其类库不是分布式的，所以把包含 4 898 431 个数据点的整个数据集拉进 R 中是不可行的，但是拉一些样本倒是很容易的：

```
clustering <- predict(kmeans_model, numeric_only)
clustering_sample <- collect(sample(clustering, FALSE, 0.01)) ❶

str(clustering_sample)

...
'data.frame': 48984 obs. of 39 variables:
 $ duration                : int 0 0 0 0 0 0 0 0 0 0 ...
 $ src_bytes               : int 181 185 162 254 282 310 212 214 181 ...
 $ dst_bytes               : int 5450 9020 4528 849 424 1981 2917 3404 ...
 $ land                    : int 0 0 0 0 0 0 0 0 0 0 ...
...
 $ prediction              : int 33 33 33 0 0 0 0 0 33 33 ...
```

❶ 按 1% 比率进行无放回采样。

`clustering_sample` 实际上是 R 的一个本地 DataFrame，而不是 Spark 的 DataFrame，所以它可以像 R 的任何其他数据一样操作。在上面的代码中，`str()` 方法显示了 DataFrame 的结构。例如，它可以提取簇群标号，然后显示簇群标号分布的统计信息：

```
clusters <- clustering_sample["prediction"] ❶
data <- data.matrix(within(clustering_sample, rm("prediction"))) ❷

table(clusters)

...
clusters
    0    11    14    18    23    25    28    30    31    33    36   ...
47294     3     1     2     2   308   105     1    27  1219    15   ...
```

❶ 只有簇群标号的列。

❷ 簇群标号的列以外的列。

例如，这表明大多数的点都属于簇群 0。尽管在 R 中可以做更多的事情，但这超出了本书的范围。

为了对数据进行可视化，需要用到一个名为 `rgl` 的库。只有在 RStudio 中运行这个示例时，才能使用这个功能。首先，安装（只需一次）并加载类库：

```
install.packages("rgl")
library(rgl)
```

注意，R 可能提示下载其他包或编译工具以完成安装，因为安装程序包意味着从源代码编译它。

这个数据集是 38 维的，需要将数据投射到最多三维的空间上才能进行可视化，这里我们进行**随机投射**。

```
random_projection <- matrix(data = rnorm(3*ncol(data)), ncol = 3) ❶
random_projection_norm <-
  random_projection / sqrt(rowSums(random_projection*random_projection))

projected_data <- data.frame(data %*% random_projection_norm) ❷
```

❶ 随机进行三维投影并归一化。

❷ 投影并构建一个新的 DataFrame。

使用 3 个随机单位向量，R 代码将 38 维数据集向这 3 个单位向量方向进行投影，从而得到一个三维数据集。这里我们采用的是简单粗暴的降维方法。当然我们也可以采用更加复杂的降维算法，比如主成分分析算法（PCA，https://en.wikipedia.org/wiki/Principal_component_analysis）和奇异值分解算法（SVD，https://en.wikipedia.org/wiki/Singular_value_decomposition）。这些算法在 R 里都有，但运行时间很长。对于本章示例数据的可视化，采用随机投影方法效果差别不大但速度却要快很多。

最后通过交互式三维可视化来绘制聚类后的点：

```
num_clusters <- max(clusters)
palette <- rainbow(num_clusters)
colors = sapply(clusters, function(c) palette[c])
plot3d(projected_data, col = colors, size = 10)
```

注意，这里需要在一个支持 `rgl` 库和图形显示的环境中运行 RStudio。例如，在 macOS 上，它需要已安装苹果开发者工具 X11。

图 5-1 为可视化的结果，它显示了三维空间的数据点，不同簇用不同颜色表示。许多点都重叠在一起，而且结果很稀疏，因此有些难懂。然而图形明显呈“L”状。看来数据点在两个维度上有变化，而在其他维度上没什么变化。

这种现象是合理的，因为数据集有两个特征的取值范围要比其他特征大得多。大多数特征的取值范围为 0~1，但发送字节数和接收字节数这两个特征的取值为 0 到数万字节。因此点的欧氏距离几乎完全由这两个特征决定，其他特征似乎根本就不存在。必须对不同的数据取值范围进行归范化，才能把特征放在近似的基准上。

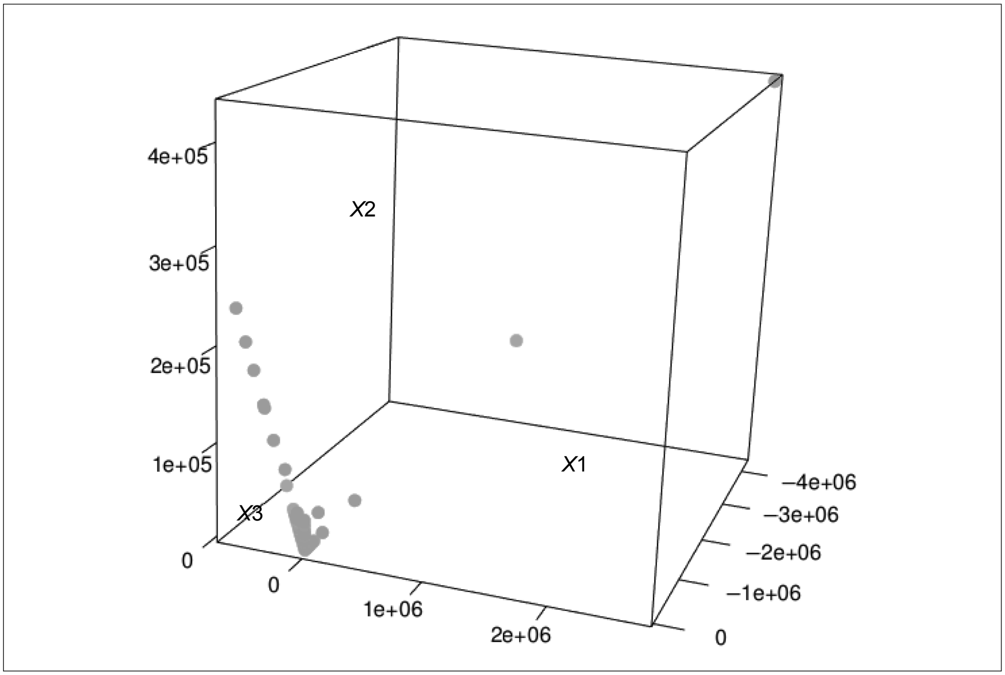


图 5-1：随机三维投影

5.8　特征的规范化

特征的规范化可以通过将每个特征转换为标准得分（https://en.wikipedia.org/wiki/Standard_score）来完成。这就是说用对每个特征值求平均，用每个特征值减去平均值，然后除以特征值的标准差，如下标准分计算公式所示：

$$\text{normalized}_i = \frac{\text{feature}_i - \mu_i}{\sigma_i}$$

由于减去平均值相当于把所有数据点沿相同方法移动相同距离，不影响点之间的欧氏距离，所以实际上减去平均值对聚类结果没有影响。

MLlib 提供了 `StandardScaler` 组件，用于特征的规范化，并且能很轻松地加入集群的管道中。

增加 *k* 的取值范围并在规范化的数据上运行相同测试：

```
import org.apache.spark.ml.feature.StandardScaler

def clusteringScore2(data: DataFrame, k: Int): Double = {
  val assembler = new VectorAssembler().
```

```
    setInputCols(data.columns.filter(_ != "label")).
    setOutputCol("featureVector")
  val scaler = new StandardScaler()
    .setInputCol("featureVector")
    .setOutputCol("scaledFeatureVector")
    .setWithStd(true)
    .setWithMean(false)
  val kmeans = new KMeans().
    setSeed(Random.nextLong()).
    setK(k).
    setPredictionCol("cluster").
    setFeaturesCol("scaledFeatureVector").
    setMaxIter(40).
    setTol(1.0e-5)
  val pipeline = new Pipeline().setStages(Array(assembler, scaler, kmeans))
  val pipelineModel = pipeline.fit(data)
  val kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
  kmeansModel.computeCost(pipelineModel.transform(data)) / data.count()
}

(60 to 270 by 30).map(k => (k, clusteringScore2(numericOnly, k))).
  foreach(println)
```

这有助于将维度放到更平等的基准上，而且在绝对的意义上，看点之间的绝对距离（也就是代价）要小得多。然而，*k* 值还没有出现一个明显的点，超过该点后，增加 *k* 值对于改善代价没有明显的作用：

```
(60,1.2454250178069293)
(90,0.7767730051608682)
(120,0.5070473497003614)
(150,0.4077081720067704)
(180,0.3344486714980788)
(210,0.276237617334138)
(240,0.24571877339169032)
(270,0.21818167354866858)
```

对规范化数据再次在三维空间上进行可视化。如期望的那样，图形显示出更丰富的结构。有些点分布在一个方向上，间隔相差不远，这些点可能是数据中离散维度（比如个数）的投影。由于有 100 个簇群，我们很难从图形中看出每个点属于哪个簇。图中有一个占据多数的大簇群和有许多小簇群，小簇群显示为紧凑的子区域（图 5-2 是整个三维图形中的一部分，并经过了放大处理，所以图中有些簇没有显示出来）。图 5-2 的结果虽然没有进一步提升我们的分析结果，但它进行的完整性检查是有帮助的。

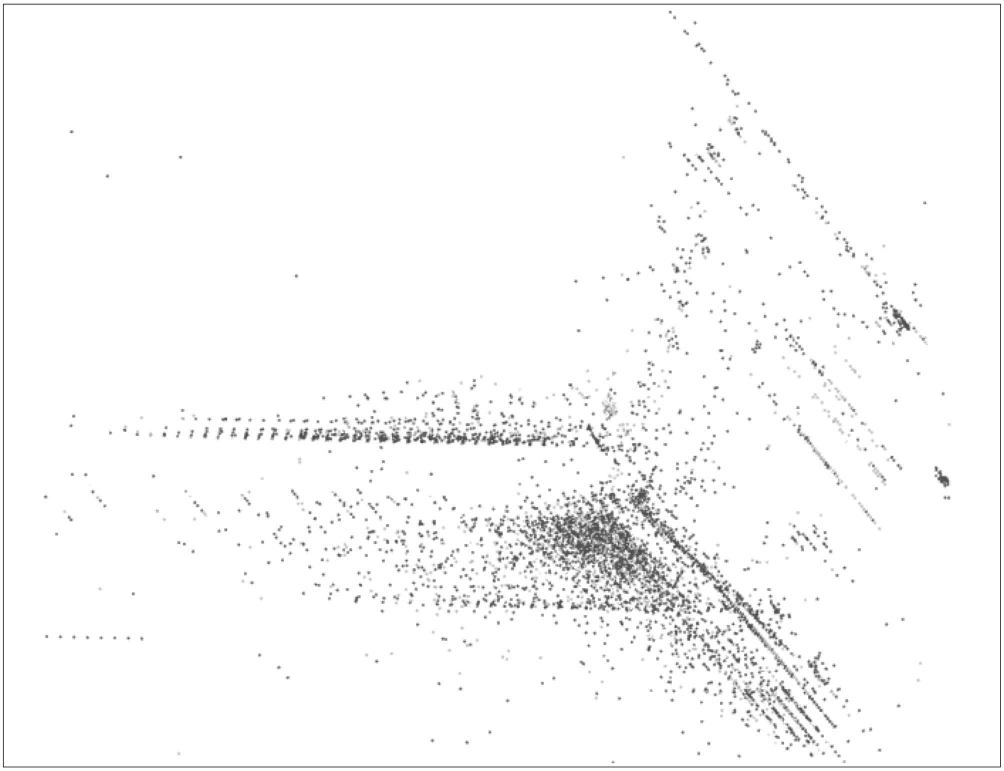


图 5-2：随机三维投影（规范化）

5.9 类别型变量

归一化使聚类结果有了可贵的进步，但聚类结果还有进一步提升的空间。比如说，几个特征由于不是数值型就被去掉了，于是这些特征里有价值的信息也被丢掉了。如果将这些信息以某种形式加回来，我们应该能得到更好的聚类。

在本章的前几节中，因为 MLlib 的 K 均值实现的欧氏距离函数中不能使用非数值型特征，所以我们把 3 个类别型特征排除掉了。这种对类别型特征的处理方法与第 4 章的不同，第 4 章将原本为类别型的特征处理成了数值型特征。

类别型特征可以用 one-hot 编码转换为几个二元特征，这几个二元特征可以看成数值型维度。举个例子，数据集的第二列代表协议类型，取值可能是 `tcp`、`udp` 或 `icmp`。可以把它们看成 3 个特征，分别取名为 `is_tcp`、`is_udp` 和 `is_icmp`。这样，特征值 `tcp` 就变成 `1,0,0`，`udp` 对应 `0,1,0`，`icmp` 对应 `0,0,1`，以此类推。

MLlib 再一次提供了实现这种转换的组件。事实上像 `protocal_type` 这样的 one-hot 编码字符串特征需要分两步骤处理。第一步，使用 `StringIndexer` 将字符串转换为整数索引，如

0、1、2 等。第二步，这些整数索引用 OneHotEncoder 编码成一个向量。这两步本身也可以当作一个小型的管道。

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

def oneHotPipeline(inputCol: String): (Pipeline, String) = {
  val indexer = new StringIndexer().
    setInputCol(inputCol).
    setOutputCol(inputCol + "_indexed")
  val encoder = new OneHotEncoder().
    setInputCol(inputCol + "_indexed").
    setOutputCol(inputCol + "_vec")
  val pipeline = new Pipeline().setStages(Array(indexer, encoder))
  (pipeline, inputCol + "_vec") ❶
}
```

❶ 返回管道和输出向量列的名称。

该方法会生成一个管道，可以作为一个组件添加到整个聚类管道中。管道是可以组装的。最后我们要做就是确保将新的输出向量列添加到 VectorAssembler 的输出中，并像以前一样进行缩放、聚类和评估。为简洁起见，此处省略源代码，但可以在本章附带的存储库中找到。

```
(60,39.739250062068685)
(90,15.814341529964691)
(120,3.5008631362395413)
(150,2.2151974068685547)
(180,1.587330730808905)
(210,1.3626704802348888)
(240,1.1202477806210747)
(270,0.9263659836264369)
```

这些样本结果表明，从 k=180 这个点开始，评分值的变化趋于平缓。至少现在聚类使用了所有的输入特征。

5.10 利用标号的熵信息

前面在对聚类质量做快速的完整性检查时，我们使用了数据点的类别标号信息。这个概念可以进一步规范化，将其作为评价聚类质量的一种可能方法，这样我们就可以用它选择 k 值。

标签告诉我们每个数据点的真实性质。好的聚类应该和人工标签保持一致，大部分情况下，标签相同的数据点应聚在一起，而标签不同的数据点不应该在一起，并且簇内的数据点标签相同。

第 4 章我们定义了同质的度量指标：Gini 不纯度和熵。这两个指标都是每个簇中标签比例

的函数，当比例偏斜为少数标签甚至是一个标签时，其值会变得很小。这里用熵来说明。

```
def entropy(counts: Iterable[Int]): Double = {
  val values = counts.filter(_ > 0)
  val n = values.map(_.toDouble).sum
  values.map { v =>
    val p = v / n
    -p * math.log(p)
  }.sum
}
```

良好的聚类结果簇中样本类别大体相同，因而熵值较低。我们可以对各个簇的熵加权平均，将结果作为聚类得分：

```
val clusterLabel = pipelineModel.transform(data).
  select("cluster", "label").as[(Int, String)] ❶

val weightedClusterEntropy = clusterLabel.
  groupByKey { case (cluster, _) => cluster }. ❷
  mapGroups { case (_, clusterLabels) =>
    val labels = clusterLabels.map { case (_, label) => label }.toSeq
    val labelCounts = labels.groupBy(identity).values.map(_.size) ❸
    labels.size * entropy(labelCounts)
  }.collect()

weightedClusterEntropy.sum / data.count() ❹
```

❶ 对每个数据预测簇类别。

❷ 按簇提取标号集合。

❸ 计算集合中各簇标号出现的次数。

❹ 根据簇大小计算熵的加权平均。

跟以前一样，可以根据上面的分析结果大致看出 *k* 的合适取值。随着 *k* 的增加，熵不一定会减小，因此我们找到的可能是一个局部最小值。这里结果同样表明，*k* 取 180 可能比较合理，因为它的得分实际上低于 150 **以及** 210：

```
(60,0.03475331900669869)
(90,0.051512668026335535)
(120,0.02020028911919293)
(150,0.019962563512905682)
(180,0.01110240886325257)
(210,0.01259738444250231)
(240,0.01357435960663116)
(270,0.010119881917660544)
```

5.11 聚类实战

现在我们对聚类模型质量比较有信心了。最后我们来对规范化后的全体数据进行聚类，并

取 k=180。为了大致了解聚类结果，这里我们同样把每个簇的标号打印出来。聚类的结果中大部分属于同一簇，以及其他的少部分簇。

```
val pipelineModel = fitPipeline4(data, 180) ❶
val countByClusterLabel = pipelineModel.transform(data).
  select("cluster", "label").
  groupBy("cluster", "label").count().
  orderBy("cluster", "label")
countByClusterLabel.show()

...
+-------+----------+------+
|cluster|     label| count|
+-------+----------+------+
|      0|     back.|   324|
|      0|   normal.| 42921|
|      1|  neptune.|  1039|
|      1|portsweep.|     9|
|      1|    satan.|     2|
|      2|  neptune.|365375|
|      2|portsweep.|   141|
|      3|portsweep.|     2|
|      3|    satan.| 10627|
|      4|  neptune.|  1033|
|      4|portsweep.|     6|
|      4|    satan.|     1|
...
```

❶ 要了解 `fitPipline4()` 的定义，请参见附带的源代码。

现在可以建立一个真正的异常检测系统了。异常检测时需要度量新数据点到最近的簇质心的距离。如果这个距离超过某个阈值，那么就表示这个新数据点是异常的。我们可以把阈值设为已知数据中离中心最远的第 100 个点到中心的距离。

```
import org.apache.spark.ml.linalg.{Vector, Vectors}

val kMeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
val centroids = kMeansModel.clusterCenters

val clustered = pipelineModel.transform(data)
val threshold = clustered.
  select("cluster", "scaledFeatureVector").as[(Int, Vector)].
  map { case (cluster, vec) => Vectors.sqdist(centroids(cluster), vec) }.
  orderBy($"value".desc).take(100).last ❶
```

❶ 单个输出，隐式命名为“value”。

最后一步就是在新数据点出现的时候使用阈值进行评估。举个例子，可以用 Spark Streaming 对来源于 Flume、Kafka 或 HDFS 文件的小批量数据计算函数值。只要计算结果超过阈值就触发邮件报警或更新数据库。

作为示例，我们在原始数据集上进行异常检查。这样就能找出输入数据中我们认为最不寻常的异常数据。

```
val originalCols = data.columns
val anomalies = clustered.filter { row =>
  val cluster = row.getAs[Int]("cluster")
  val vec = row.getAs[Vector]("scaledFeatureVector")
  Vectors.sqdist(centroids(cluster), vec) >= threshold
}.select(originalCols.head, originalCols.tail:_*) ❶

anomalies.first() ❷

...
[9,tcp,telnet,SF,307,2374,0,0,1,0,0,1,0,1,0,1,3,1,0,0,0,0,1,1,
 0.0,0.0,0.0,0.0,1.0,0.0,0.0,69,4,0.03,0.04,0.01,0.75,0.0,0.0,
 0.0,0.0,normal.]
```

❶ 注意（String, String*）这一选择列的奇怪签名语法。

❷ show() 方法也可以工作，但不好理解了。

这个例子展示了对 DataFrame 稍微不同的操作方式。纯 SQL 不能表达平方距离的计算。如前所述，可以使用 UDF 来定义返回两个列之间平方距离的函数。但是，也可以采用编程方式使用 Row 对象与数据中的每一行进行交互，就像在 JDBC 中一样。

网络安全专家估计很快就能看出为什么这是一个异常连接或者其实不是一个异常连接。这个数据点被标记为 normal，但需要连接到 69 台不同的主机。

5.12 小结

本质上，KMeansModel 模型本身就是一个异常检测系统。前面我们在代码中演示了如何用它来检测数据中的异常。这些代码同样可以用于 Spark Streaming（https://spark.apache.org/streaming/）中，在数据出现时以准实时的方式对数据评分，如果有异常就触发报警或审查。

MLlib 还有 KMeansModel 的一个变体，被称为 StreamingKMeans。StreamingKMeans 模型能够根据增量对簇进行更新。有了 StreamingKMeans，我们就不再只是用已知簇群评价新数据，而是进一步做到近似地学习新数据如何影响聚类过程了。这也可以集成到 Spark Streaming 上。然而，它还没有对照新的 DataFrame API 进行更新。

这个聚类模型只是一个简单的实现。比如由于 Spark MLlib 目前只提供了欧氏距离一种实现，所以本章示例中我们采用的是欧氏距离。Spark MLlib 今后版本可能会提供其他距离函数，比如马氏距离（https://en.wikipedia.org/wiki/Mahalanobis_distance），这样就可能会更好地描述特征的关联关系。

我们还可以用更复杂的聚类质量评估指标（https://en.wikipedia.org/wiki/Cluster_analysis#Internal_evaluation），比如轮廓系数（Silhouette coefficient，https://en.wikipedia.org/wiki/Silhouette_(clustering)），来选择合适的 *k* 值。这些指标既可以评价簇内点的紧密程度又可以评价点与其他簇之间的紧密程度。最后，除了 K 均值聚类外，我们还可以尝试其他模型。比如，高斯混合模型（https://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model）或 DBSCAN（https://en.wikipedia.org/wiki/DBSCAN）可用于处理数据点和簇中心之间更加微妙的关系。Spark MLlib 已经实现了高斯混合模型（http://spark.apache.org/docs/latest/ml-clustering.html#gaussian-mixture-model-gmm）。Spark MLlib 的后续版本或其他基于它的库可能会提供这些模型的实现。

当然，聚类的应用不只是局限在异常检查。事实上聚类往往与使用场景相关。在这些场景中，实际的簇很重要。比如，可以根据用户的行为、偏好和属性来聚类。每个簇本身就可能代表一类顾客，将这类顾客区分出来是很有意义的。相比学习“20~34 岁”和“女性”之类的普通群组划分，这种客户划分方法的数据驱动程度更高。

第 6 章

基于潜在语义分析算法分析维基百科

作者：桑迪·里扎

去年的斯诺登夫妇如今在何方？

——Capt. Yossarian

数据工程的大多数任务都是把数据“组装”为某种可查询的格式。我们可以用形式化语言对结构化的数据进行查询。例如，用 SQL 来查询结构化的表数据。虽然访问表数据实际上也非易事，但从表面上看，准备这种数据还是很直观的：只不过是从多个数据源读取数据并写入一张表，然后可能再进行数据清洗或智能数据融合。相比之下，处理非结构化的文本数据则要困难得多。要将文本数据处理成人类可以理解的格式可不是“组装”这么简单。即使情况简单，也需要给它建立索引；如果情况复杂，甚至还得做某种“强制性”的工作。给包含某个特定词汇的文档创建标准索引可以加快查询的速度。但有时候我们希望找出文档中是否有包含某个特定词汇的相关概念，而不是仅仅匹配这个特定词汇的字符串。标准索引常常无法发掘文本主题的潜在结构。

潜在语义分析（latent semantic analysis，LSA）是一种自然语言处理和信息检索技术，其目的是更好地理解文档语料库以及文档中词项的关系。它将语料库提炼成一组相关**概念**，每个概念捕捉了数据中一个不同的主题，且通常与语料库讨论的主题相符合。为了不涉及过多数学细节，我们可以把每个概念看成由 3 个属性组成：语料库中文档的相关度、语料库中词项的相关度，以及概念对描述主题的重要性评分。举个例子，LSA 可能发现一个概念与“Asimov”和“robot”高度相关，与文档“foundation series”和“science fiction”也高

度相关。通过挑选出最重要的概念，LSA 可以过滤掉不相关的噪声，合并同时出现的主题，从而简化数据。

这种化简技术应用得非常广泛。它可以计算词项与词项、文档与文档、词项与文档之间的相似度评分。通过发掘语料库中的不同主题模式，LSA 算法在计算相似度评分时不再简单地基于词项出现的频率和两个词项同时出现的频率，而是基于更深入的分析。这些相似度度量方法适合根据词项查询相关文档、按主题将文档分组和找到相关的词项等任务。

LSA 在降维过程中使用一种称为**奇异值分解**（SVD）的线性代数技术。我们可以把 SVD 看成第 3 章讨论的 ALS 分解的升级版。首先，需要根据词项在每个文档中的出现次数构造一个**文档 – 词项**矩阵。矩阵中每个文档对应一列，每个词项对应一行，矩阵的每个元素代表某个词项在对应文档中的重要性。接着，SVD 将矩阵分解成三个矩阵，其中一个矩阵代表文档中出现的概念，另一个代表词项对应的概念，还有一个代表每个概念的重要度。这三个矩阵的结构可以让我们通过去掉最不重要的概念所对应的行和列而获得原始矩阵的一个**低阶近似**。也就是说，将这些低阶近似矩阵相乘可以得到原始矩阵的近似，去掉的概念越多，就越失真。

本章将基于人类知识库的潜在语义关系讨论如何构建人类知识库的查询模型。具体来说，我们将在维基百科所有文档组成的语料库上运行 LSA 算法，文本格式的语料库大小约为 46 GB。我们会讨论如何使用 Spark 进行数据预处理，包括数据读取、清洗并转换成数值。我们会演示如何计算 SVD，并解释怎样理解和利用 SVD 算法。

除了 LSA 之外，SVD 还存在其他很多用途，比如识别气候趋势（如著名的 Michael Mann“曲棍球杆”曲线：https://en.wikipedia.org/wiki/Hockey_stick_controversy）、人脸识别和图像压缩等。Spark 的实现可以在大量数据集上执行矩阵因子分解，这就将该新技术推向了全新的应用领域。

6.1 文档–词项矩阵

在进行分析之前，LSA 算法需要将语料库中的文本转换成文档 – 词项矩阵。该矩阵的每列代表语料库中出现的一个词项，每行代表一篇文档。不严格地讲，矩阵中每个元素值代表了相应列上的词项相对于相应行上的文档的权重。人们提出了几种方法来表示这种权重，其中用得最多的是用**词项频率**除以**文档频率**，该方法通常简写为 TF-IDF（term frequency times inverse document frequency）。下面是这个公式的 Scala 代码版本。我们实际上并不会使用这段代码，因为 Spark 提供了自己的实现。

```
def termDocWeight(termFrequencyInDoc: Int, totalTermsInDoc: Int,
    termFreqInCorpus: Int, totalDocs: Int): Double = {
  val tf = termFrequencyInDoc.toDouble / totalTermsInDoc
  val docFreq = totalDocs.toDouble / termFreqInCorpus
```

```
    val idf = math.log(docFreq)
    tf * idf
  }
```

TF-IDF 体现了我们对词项与文档关联度的两个直观理解。第一，一个词项在文档中出现的次数越多，它相对于文档的重要性越高。第二，总体上讲词项是不平等的。文档中出现语料库中罕见词项的意义比出现常见词项更大，因此指标就是词项在所有语料库中出现次数的倒数。

词项在语料库中的频率分布往往呈指数型。一个常用词出现的次数往往是一个次常用词出现次数的十几倍，是一个罕见词出现次数的一百多倍。如果计算指标时直接除以原始文档频率，罕见词的权重就会过大，这时相比之下其他非罕见词的权重几乎可以忽略不计了。为了刻画这种指数分布，算法对逆文档频率取对数。这样文档频率的差别就从乘数级变成了加数级。

这个模型依赖几个假设。它把每个文档看成词项的集合，也就是说算法没有考虑词项的顺序、句式结构或否定情况。由于只针对每个词项，模型很难处理**多义词**。举个例子，“Radiohead is the best band ever”和“I broke a rubber band”这两句话中的词项“band”含义不同，模型对此就不能区分。如果两个句子在语料库中出现的次数相同，模型可能就会把 Radiohead 和 rubber 关联在一起。

本章所用的语料库有 1000 万个文档。如果把那些晦涩的技术术语包含在内，英语语言总共约有 100 万个词项。本章的语料库只包含其中的大约几万个。因为语料库的文档数远远大于词项数，所以我们的文档－词项矩阵应该是行矩阵，由许多稀疏向量组成，每个向量代表一个文档。

将原始的维基百科导出文件转成文档－词项矩阵需要进行许多预处理。首先，输入是一个巨大的 XML 文件，其中每个文档由 `<page>` 标签分隔。我们需要先对输入进行拆分，然后才能将维基百科格式转换成纯文本格式。接着纯文本被拆成词条（token）。将词条的不同曲折词缀还原为词根的过程称为**词形归并**（lemmatization）。经过词形归并之后得到的词条可以用于计算词项频率和文档频率。最后我们将这些频率组织成需要的向量对象。在本书的代码库中，执行这些步骤的所有代码都封装在 `AssembleDocumentTermMatrix` 类中。

预处理的前几步都可以根据文档进行完全并行化（对应 Spark 中的一组 `map` 函数），但计算逆文档频率时需要对所有文档进行汇总。可以利用许多通用的 NLP 工具和维基百科特有的提取工具来完成这些步骤。

6.2 获取数据

我们可以从维基百科上导出所有文章，导出的文件是一个巨大的 XML 文件。我们先从 https://dumps.wikimedia.org/enwiki 下载这个 XML 文件，然后将这个文件写入 HDFS，示例

代码如下：

```
$ curl -s -L https://dumps.wikimedia.org/enwiki/latest/\
$ enwiki-latest-pages-articles-multistream.xml.bz2 \
$   | bzip2 -cd \
$   | hadoop fs -put - wikidump.xml
```

这个过程要花一段时间。

最好可以使用包含几个节点的小集群来处理这种体量的数据。如果在本地机器上运行本章的代码，使用维基百科的导出页面功能（https://en.wikipedia.org/wiki/Special:Export）生成一个较小的转储是一个更好的选择。尝试从一个有很多页面和几个子类目的类目，比如 Megafauna 和 Geometry，下载所有页面。如果要运行以下代码，请将转储下载到 ch06-lsa/ 目录下，并重命名为 wikidump.xml。

6.3 分析和准备数据

下面是导出文件的开头部分：

```
    <page>
      <title>Anarchism</title>
      <ns>0</ns>
      <id>12</id>
      <revision>
        <id>584215651</id>
        <parentid>584213644</parentid>
        <timestamp>2013-12-02T15:14:01Z</timestamp>
        <contributor>
          <username>AnomieBOT</username>
          <id>7611264</id>
        </contributor>
        <comment>Rescuing orphaned refs (&quot;autogenerated1&quot; from rev
        584155010; &quot;bbc&quot; from rev 584155010)</comment>
        <text xml:space="preserve">{{Redirect|Anarchist|the fictional character|
        Anarchist (comics)}}
{{Redirect|Anarchists}}
{{pp-move-indef}}
{{Anarchism sidebar}}

'''Anarchism''' is a [[political philosophy]] that advocates [[stateless society|
stateless societies]] often defined as [[self-governance|self-governed]]
voluntary institutions,&lt;ref&gt;&quot;ANARCHISM, a social philosophy that
rejects authoritarian government and maintains that voluntary institutions are
best suited to express man's natural social tendencies.&quot; George Woodcock.
&quot;Anarchism&quot; at The Encyclopedia of Philosophy&lt;/ref&gt;&lt;ref&gt;
&quot;In a society developed on these lines, the voluntary associations which
already now begin to cover all the fields of human activity would take a still
greater extension so as to substitute
...
```

现在打开 Spark shell。为了简化工作，我们使用几个工具包。GitHub 上有一个 Maven 项目，我们可以用它来生成一个 JAR 文件，这个 JAR 文件包含所有这些依赖包：

```
$ cd ch06-lsa/
$ mvn package
$ spark-shell --jars target/ch06-lsa-2.0.0-jar-with-dependencies.jar
```

在开始处理这个转储之前，我们需要去掉格式并得到内容。为了方便后续处理，需要先生成一个（标题、文档内容）二元组的数据集。Cloud9 项目提供了一组十分实用的 API 来完成这个任务。

Cloud9 根据 Apache Mahout 项目编写了一个 `XMLInputFormat` 类，这个类可以把巨大的维基百科导出文件拆成文档。现在我们用这个类来创建一个数据集：

```
import edu.umd.cloud9.collection.XMLInputFormat
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.io._

val path = "wikidump.xml"
@transient val conf = new Configuration()
conf.set(XMLInputFormat.START_TAG_KEY, "<page>")
conf.set(XMLInputFormat.END_TAG_KEY, "</page>")
val kvs = spark.sparkContext.newAPIHadoopFile(path, classOf[XMLInputFormat],
  classOf[LongWritable], classOf[Text], conf)
val rawXmls = kvs.map(_._2.toString).toDS()
```

要讲述如何将维基百科的 XML 文件转成纯文本估计需要整整一章。幸运的是，我们可以利用 Cloud9 项目提供的 API 来完成所有工作：

```
import edu.umd.cloud9.collection.wikipedia.language._
import edu.umd.cloud9.collection.wikipedia._

def wikiXmlToPlainText(pageXml: String): Option[(String, String)] = {
  // 在从Cloud9编写完成后，维基百科对它们的转储进行了一点点更新
  // 所以有时需要用这个替换的小技巧来完成解析工作
  val hackedPageXml = pageXml.replaceFirst(
    "<text xml:space=\"preserve\" bytes=\"\\d+\">",
    "<text xml:space=\"preserve\">")

  val page = new EnglishWikipediaPage()
  WikipediaPage.readPage(page, hackedPageXml)
  if (page.isEmpty) None
  else Some((page.getTitle, page.getContent))
}

val docTexts = rawXmls.filter(_ != null).flatMap(wikiXmlToPlainText)
```

6.4　词形归并

现在我们得到了纯文本形式的语料库，接下来要提炼出一组词项。这一步有几点需要特别注意。第一，像 the 和 is 之类的常用词不会为模型提供有用信息，却占用了大量空间。去掉这些**停用词**（stop word）不但能节省空间，还能提高忠实度。第二，相同意思的词项可能有不同词形。比如，monkey 和 monkeys 不应该算成不同词项，再比如 nationalize 和 nationalization。把这些不同屈折词缀合并成单个词项的过程称为**词干还原**（stemming）或**词形归并**（lemmatization）。词干还原指去除单词两端词缀的启发式技术，词形归并方法则更加规则化。比如前者可能将 drew 截断为 dr，而后者则更可能给出 draw 这个正确结果。Stanford Core NLP 项目是一个非常优秀的词干规约工具，它提供了 Java API，我们可以在 Scala 中调用。

下面的代码接收纯文本形式的文档数据集，对文档进行词形归并，过滤掉其中的停用词。请注意，此代码依赖于一个名为 stopwords.txt 的停用词文件，这个文件放在附带的源代码库中，应提前下载到当前目录下：

```
import scala.collection.JavaConverters._
import scala.collection.mutable.ArrayBuffer
import edu.stanford.nlp.pipeline._
import edu.stanford.nlp.ling.CoreAnnotations._
import java.util.Properties
import org.apache.spark.sql.Dataset

def createNLPPipeline(): StanfordCoreNLP = {
  val props = new Properties()
  props.put("annotators", "tokenize, ssplit, pos, lemma")
  new StanfordCoreNLP(props)
}

def isOnlyLetters(str: String): Boolean = {
  str.forall(c => Character.isLetter(c))
}

def plainTextToLemmas(text: String, stopWords: Set[String],
    pipeline: StanfordCoreNLP): Seq[String] = {
  val doc = new Annotation(text)
  pipeline.annotate(doc)

  val lemmas = new ArrayBuffer[String]()
  val sentences = doc.get(classOf[SentencesAnnotation])
  for (sentence <- sentences.asScala;
       token <- sentence.get(classOf[TokensAnnotation]).asScala) {
    val lemma = token.get(classOf[LemmaAnnotation])
    if (lemma.length > 2 && !stopWords.contains(lemma)
        && isOnlyLetters(lemma)) { ❶
      lemmas += lemma.toLowerCase
    }
  }
```

```
    lemmas
  }

  val stopWords = scala.io.Source.fromFile("stopwords.txt").getLines().toSet
  val bStopWords = spark.sparkContext.broadcast(stopWords) ❷

  val terms: Dataset[(String, Seq[String])] =
    docTexts.mapPartitions { iter =>
      val pipeline = createNLPPipeline()
      iter.map { case(title, contents) =>
        (title, plainTextToLemmas(contents, bStopWords.value, pipeline))
      }
    } ❸
```

❶ 为剔除垃圾词元，需对词元设定最低要求。

❷ 广播停用词可以节省 `executor` 的内存空间。

❸ 这里使用了 `mapPartitions`，对每个分区只初始化一个 NLP 管道对象，而不是为每一个文档初始化一个 NLP 管道对象。

6.5 计算TF-IDF

现在 `terms` 指向一个词项序列数据集，每个序列对应一个文档。下一步我们要计算每个词项在每个文档和整个语料库中的频率。spark.ml 包中包含了计算 TF-IDF 的 `Estimator` 和 `Transformer` 的实现。

为了使用这些 `Estimator` 和 `Transformer`，首先需要将数据集转换成 DataFrame：

```
val termsDF = terms.toDF("title", "terms")
```

过滤掉没有词或者只有一个词的文档：

```
val filtered = termsDF.where(size($"terms") > 1)
```

`CountVectorizer` 是一个计算词频的 `Estimator`。`CountVectorizer` 扫描数据来建立一张词汇表。该词汇表其实就是一个整数到词项的映射，这个词汇表封装到 `CountVectorizerModel` 这个 `Transformer` 中。然后可以使用 `CountVectorizerModel` 为每个文档生成一个词频的向量。词汇表中每个词项在该向量中都有一个组成部分，每个组成部分的值是词汇在文档中出现的次数。Spark 在这里使用稀疏向量，因为每个文档通常只会用到整个词汇表中的一小部分。

```
import org.apache.spark.ml.feature.CountVectorizer

val numTerms = 20000
val countVectorizer = new CountVectorizer().
  setInputCol("terms").setOutputCol("termFreqs").
  setVocabSize(numTerms)
```

```
val vocabModel = countVectorizer.fit(filtered)
val docTermFreqs = vocabModel.transform(filtered)
```

请注意 `setVocabSize` 的用法。这个语料库包含了几百万个词项，但很多都是非常专业的单词，仅出现在一两个文档中。过滤使用频率低的词项可以提高性能并消除噪声。当我们在 `Estimator` 上设置词典大小时，它会保留最常使用的单词。

得到的 DataFrame 在后面的章节中将至少用到两次：一次用于计算逆文档频率，一次用于计算最终文档－词项的矩阵。所以，将它缓存起来是个不错的做法：

```
docTermFreqs.cache()
```

现在有了文档频率，接下来计算逆文档频率。此处使用 `IDF`，这又是一个 `Estimator`，我们用它来计算每个词项在语料库中出现的次数，然后使用这些计数来计算每个词项的 IDF 比例因子。它产生的 `IDFModel` 可以将这些比例因子应用于数据集中每个向量的每一个词项。

```
import org.apache.spark.ml.feature.IDF

val idf = new IDF().setInputCol("termFreqs").setOutputCol("tfidfVec")
val idfModel = idf.fit(docTermFreqs)
val docTermMatrix = idfModel.transform(docTermFreqs).select("title", "tfidfVec")
```

当把数据从 DataFrame 转换成向量和矩阵的那一刻起，我们就失去了通过字符串访问数据的能力。因此，如果想要将我们所学到的内容与可识别的实体联系起来，那么将矩阵中的位置映射到原始语料库中的词项和文档标题是十分重要的。词向量中的位置等同于文档－词项矩阵中的列。这些位置到词项字符串的映射已经保存在我们的 `CountVectorizerModel` 中。我们可以通过如下方式访问它：

```
val termIds: Array[String] = vocabModel.vocabulary
```

创建一个行 ID 到文档标题的映射要更复杂一些，我们可以使用 `zipWithUniqueId` 方法获取这个映射，它将 DataFrame 中的每一行与一个确定的唯一 ID 关联起来。我们的操作基于以下基本事实：对 DataFrame 做某种转换，然后调用 `zipWithUniqueId` 函数，转换后的行的唯一 ID 保持不变，只要所做的转换不改变 DataFrame 的行数和分区。因此我们可以把变换后的行和 DataFrame 中的行 ID 关联起来，这样就能关联到文档标题了：

```
val docIds = docTermFreqs.rdd.map(_.getString(0)).
  zipWithUniqueId().
  map(_.swap).
  collect().toMap
```

6.6 奇异值分解

现在有了文档－词项矩阵 M，我们的分析工作就可以进入矩阵分解和降维的步骤了。

MLlib 提供了一个奇异值分解算法（SVD）的实现，能处理巨型矩阵。奇异值分解算法接受一个 $m \times n$ 维矩阵，返回 3 个矩阵。这 3 个矩阵的乘积近似等于原始的 $m \times n$ 维矩阵：

$$M \approx USV^{\mathrm{T}}$$

矩阵列举如下。

- U 为 $m \times k$ 维矩阵，U 中的列是文档空间的正交基。
- S 为 $k \times k$ 型对角阵，每个对角元素代表一个概念的强度。
- V 为 $k \times n$ 型矩阵，V 中的列是词项空间的正交基。

对于 LSA 模型，m 是文档的个数，n 是词项的个数。分解过程有一个参数 k，其值不大于 n，代表要保留的概念的个数。当 $k=n$ 时，分解矩阵的乘积精确重构出原始矩阵。$k<n$ 时，分解矩阵的乘积是原始矩阵的一个低阶近似。一般 k 的取值要远小于 n。SVD 算法保证在最多只采用 k 个概念的约束下，算法结果是对原始矩阵的最优逼近（由 L2 范式定义，也就是误差平方和最小）。

在撰写本文时，基于 DataFrame 的 spark.ml 包中没有 SVD 的实现。但是，基于 RDD 的旧版 spark.mllib 中却有。这意味着为了计算文档－词项矩阵，我们需要将 DataFrame 转换成 `RDD[Vector]`。最重要的是，spark.ml 和 spark.mllib 包都有自己的 `Vector` 类，之前我们使用的 spark.ml 的 `Transformer` 生成的是 spark.ml 中的 `Vector` 对象，但 SVD 实现只接受 spark.mllib 中的 `Vector` 对象，所以需要进行转换。虽然这段代码不够优雅，但是它可以达成目的：

```
import org.apache.spark.mllib.linalg.{Vectors,
  Vector => MLLibVector}
import org.apache.spark.ml.linalg.{Vector => MLVector}


val vecRdd = docTermMatrix.select("tfidfVec").rdd.map { row =>
  Vectors.fromML(row.getAs[MLVector]("tfidfVec"))
}
```

要得到矩阵的奇异值分解，只要将行向量 RDD 包装为 `RowMatrix` 并调用 `computeSVD` 即可，代码如下：

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix

vecRdd.cache()
val mat = new RowMatrix(vecRdd)
val k = 1000
val svd = mat.computeSVD(k, computeU=true)
```

由于计算过程需要多次使用 RDD 数据，所以我们应该事先将 RDD 缓存起来。驱动程序端运算的空间复杂度为 $O(nk)$，每个任务的空间复杂度为 $O(n)$，需要使用数据的次数为 $O(k)$。

注意，**词项空间**中的每个向量都有一个词项权重，**文档空间**中的每个向量都有一个文档权

重，**概念空间**中的每个向量都有一个概念权重。每个词项、文档或概念都在各自空间中定义了一个**轴**，词项、文档和概念的权重就是在轴方向上的长度。每个词项或文档向量都可以映射为概念空间里的相应向量。每个概念向量可能对应多个词向量和文档向量，其中包括一个规范化词向量和文档向量，对概念向量进行逆向转换就得到规范化的词向量和文档向量。

V 是 $n\times k$ 型矩阵，每一行对应一个词项，每一列对应一个概念。这个矩阵定义了词项空间到概念空间的映射。其中，词项空间中每个点是一个 n 维向量，向量的每个元素是每个词项的权重；概念空间中每个点是一个 k 维向量，向量的每个元素是每个概念的权重。

类似地，U 是 $m\times k$ 型矩阵，U 中每一行对应一个文档，每一列对应一个概念。U 定义了一个文档空间到概念空间的映射。

S 是 $k\times k$ 型对角阵，其中保存了奇异值。S 中每个对角线上的元素对应了一个概念（因此对应了 V 和 U 中的一列）。奇异值的大小对应了概念的重要程度，亦即概念在解释不同主题时的能力。SVD 的一种可能但效率不高的实现是先得到 k 阶分解，具体做法是先进行 n 阶分解，不停地去掉 $n-k$ 个最小奇异值，直到只剩下 k 个奇异值（当然还有 U 和 V 中对应的列）。LSA 算法的一个要点是概念中只有一小部分对表示数据是重要的。S 矩阵中的元素直接表示每个概念的重要性，它们正好是 MM^T 的特征值（eigenvalue，https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors）的平方根。

6.7 找出重要的概念

SVD 算法的输出是一组数值。我们该如何验证这些数值实际有没有作用呢？ V 矩阵表示了词项对概念的重要程度。如前所述，每个概念都对应 V 中一列，每个词项都对应 V 中一行。每个元素可以理解为词项相对于概念的相关度。因此我们可以用如下代码得到与那些最重要的概念最相关的词项：

```
import org.apache.spark.mllib.linalg.{Matrix,
  SingularValueDecomposition}
import org.apache.spark.mllib.linalg.distributed.RowMatrix

def topTermsInTopConcepts(
    svd: SingularValueDecomposition[RowMatrix, Matrix],
    numConcepts: Int,
    numTerms: Int, termIds: Array[String])
  : Seq[Seq[(String, Double)]] = {
  val v = svd.V
  val topTerms = new ArrayBuffer[Seq[(String, Double)]]()
  val arr = v.toArray
  for (i <- 0 until numConcepts) {
    val offs = i * v.numRows
    val termWeights = arr.slice(offs, offs + v.numRows).zipWithIndex
    val sorted = termWeights.sortBy(-_._1)
    topTerms += sorted.take(numTerms).map {
```

```
      case (score, id) => (termIds(id), score) ❶
    }
  }
  topTerms
}
```

❶ 最后一步找到词项向量中的位置对应的真实词项。回顾一下，termIds 是一个整数到词项的映射，可以从 CountVectorizer 获取。

注意，V 是驱动程序进程内存里的一个本地矩阵，以非分布式的计算方式得到。类似地，我们可以通过 U 得到和重要概念相关的词项，但由于 U 是一个分布式矩阵，所以代码稍有不同。

```
def topDocsInTopConcepts(
    svd: SingularValueDecomposition[RowMatrix, Matrix],
    numConcepts: Int, numDocs: Int, docIds: Map[Long, String])
  : Seq[Seq[(String, Double)]] = {
  val u  = svd.U
  val topDocs = new ArrayBuffer[Seq[(String, Double)]]()
  for (i <- 0 until numConcepts) {
    val docWeights = u.rows.map(_.toArray(i)).zipWithUniqueId() ❶
    topDocs += docWeights.top(numDocs).map {
      case (score, id) => (docIds(id), score)
    }
  }
  topDocs
}
```

❶ 上一节讨论了 monotonically_increasing_id/zipWithUniqueId 的技巧，这使得我们能够保持矩阵中的每一行，以及矩阵派生的 DataFrame 中的每一行之间的连续性，其中也包括标题的连续性。

现在我们来看看前面的一些概念：

```
val topConceptTerms = topTermsInTopConcepts(svd, 4, 10, termIds)
val topConceptDocs = topDocsInTopConcepts(svd, 4, 10, docIds)
for ((terms, docs) <- topConceptTerms.zip(topConceptDocs)) {
  println("Concept terms: " + terms.map(_._1).mkString(", "))
  println("Concept docs: " + docs.map(_._1).mkString(", "))
  println()
}

Concept terms: summary, licensing, fur, logo, album, cover, rationale,
  gif, use, fair
Concept docs: File:Gladys-in-grammarland-cover-1897.png,
  File:Gladys-in-grammarland-cover-2010.png, File:1942ukrpoljudeakt4.jpg,
  File:Σακελλαρίδης.jpg, File:Baghdad-texas.jpg, File:Realistic.jpeg,
  File:DuplicateBoy.jpg, File:Garbo-the-spy.jpg, File:Joysagar.jpg,
  File:RizalHighSchoollogo.jpg

Concept terms: disambiguation, william, james, john, iran, australis,
```

```
  township, charles, robert, river
Concept docs: G. australis (disambiguation), F. australis (disambiguation),
  U. australis (disambiguation), L. maritima (disambiguation),
  G. maritima (disambiguation), F. japonica (disambiguation),
  P. japonica (disambiguation), Velo (disambiguation),
  Silencio (disambiguation), TVT (disambiguation)

Concept terms: licensing, disambiguation, australis, maritima, rawal,
  upington, tallulah, chf, satyanarayana, valérie
Concept docs: File:Rethymno.jpg, File:Ladycarolinelamb.jpg,
  File:KeyAirlines.jpg, File:NavyCivValor.gif, File:Vitushka.gif,
  File:DavidViscott.jpg, File:Bigbrother13cast.jpg, File:Rawal Lake1.JPG,
  File:Upington location.jpg, File:CHF SG Viewofaltar01.JPG

Concept terms: licensing, summarysource, summaryauthor, wikipedia,
  summarypicture, summaryfrom, summaryself, rawal, chf, upington
Concept docs: File:Rethymno.jpg, File:Wristlock4.jpg, File:Meseanlol.jpg,
  File:Sarles.gif, File:SuzlonWinMills.JPG, File:Rawal Lake1.JPG,
  File:CHF SG Viewofaltar01.JPG, File:Upington location.jpg,
  File:Driftwood-cover.jpg, File:Tallulah gorge2.jpg

Concept terms: establishment, norway, country, england, spain, florida,
  chile, colorado, australia, russia
Concept docs: Category:1794 establishments in Norway,
  Category:1838 establishments in Norway,
  Category:1849 establishments in Norway,
  Category:1908 establishments in Norway,
  Category:1966 establishments in Norway,
  Category:1926 establishments in Norway,
  Category:1957 establishments in Norway,
  Template:EstcatCountry1stMillennium,
  Category:2012 establishments in Chile,
  Category:1893 establishments in Chile
```

第一个概念对应的文档看起来都是关于图片的，词项看起来都与图片属性和授权相关。第二个概念看起来是一个答疑页面。这说明维基百科导出文件除了包含维基百科上的原始文章，很可能还包含一些管理页面和讨论页面。通过检查中间阶段的输出，我们可以尽早发现这类问题。幸运的是，Cloud9 项目提供了过滤这些页面的功能。修改后的 `wikiXmlToPlainText` 方法代码如下：

```
def wikiXmlToPlainText(xml: String): Option[(String, String)] = {
  ...
  if (page.isEmpty || !page.isArticle || page.isRedirect ||
      page.getTitle.contains("(disambiguation)")) {
    None
  } else {
    Some((page.getTitle, page.getContent))
  }
}
```

在过滤后的文档集上重新运行处理过程，就会得到如下结果，它看起来比上一次的结果更加合理：

```
Concept terms: disambiguation, highway, school, airport, high, refer,
  number, squadron, list, may, division, regiment, wisconsin, channel,
  county
Concept docs: Tri-State Highway (disambiguation),
  Ocean-to-Ocean Highway (disambiguation), Highway 61 (disambiguation),
  Tri-County Airport (disambiguation), Tri-Cities Airport (disambiguation),
  Mid-Continent Airport (disambiguation), 99 Squadron (disambiguation),
  95th Squadron (disambiguation), 94 Squadron (disambiguation),
  92 Squadron (disambiguation)

Concept terms: disambiguation, nihilistic, recklessness, sullen, annealing,
  negativity, initialization, recapitulation, streetwise, pde, pounce,
  revisionism, hyperspace, sidestep, bandwagon
Concept docs: Nihilistic (disambiguation), Recklessness (disambiguation),
  Manjack (disambiguation), Wajid (disambiguation), Kopitar (disambiguation),
  Rocourt (disambiguation), QRG (disambiguation),
  Maimaicheng (disambiguation), Varen (disambiguation), Gvr (disambiguation)

Concept terms: department, commune, communes, insee, france, see, also,
  southwestern, oise, marne, moselle, manche, eure, aisne, isère
Concept docs: Communes in France, Saint-Mard, Meurthe-et-Moselle,
  Saint-Firmin, Meurthe-et-Moselle, Saint-Clément, Meurthe-et-Moselle,
  Saint-Sardos, Lot-et-Garonne, Saint-Urcisse, Lot-et-Garonne, Saint-Sernin,
  Lot-et-Garonne, Saint-Robert, Lot-et-Garonne, Saint-Léon, Lot-et-Garonne,
  Saint-Astier, Lot-et-Garonne

Concept terms: genus, species, moth, family, lepidoptera, beetle, bulbophyllum,
  snail, database, natural, find, geometridae, reference, museum, noctuidae
Concept docs: Chelonia (genus), Palea (genus), Argiope (genus), Sphingini,
  Cribrilinidae, Tahla (genus), Gigartinales, Parapodia (genus),
  Alpina (moth), Arycanda (moth)

Concept terms: province, district, municipality, census, rural, iran,
  romanize, population, infobox, azerbaijan, village, town, central,
  settlement, kerman
Concept docs: New York State Senate elections, 2012,
  New York State Senate elections, 2008,
  New York State Senate elections, 2010,
  Alabama State House of Representatives elections, 2010,
  Albergaria-a-Velha, Municipalities of Italy, Municipality of Malmö,
  Delhi Municipality, Shanghai Municipality, Göteborg Municipality

Concept terms: genus, species, district, moth, family, province, iran, rural,
  romanize, census, village, population, lepidoptera, beetle, bulbophyllum
Concept docs: Chelonia (genus), Palea (genus), Argiope (genus), Sphingini,
  Tahla (genus), Cribrilinidae, Gigartinales, Alpina (moth), Arycanda (moth),
  Arauco (moth)

Concept terms: protein, football, league, encode, gene, play, team, bear,
  season, player, club, reading, human, footballer, cup
Concept docs: Protein FAM186B, ARL6IP1, HIP1R, SGIP1, MTMR3,
  Gem-associated protein 6, Gem-associated protein 7, C2orf30, OS9 (gene),
  RP2 (gene)
```

前两个概念还是有些模糊，但其他概念看起来可以代表一些有意义的类别。第三个概念看起来像是法国的某些地方。第四个和第六个概念分别为动物和虫子的分类。第五个是关于选举、城市和政府的。第七个概念对应的文章是关于蛋白质的，但有些词项与足球相关，或许牵扯到了提升健美效果的药物？虽然每个概念都有一些让人费解的词出现，但所有概念都显示出一定程度上的主题连贯性。

6.8 基于低维近似的查询和评分

词项与文档之间的相关度如何？词项与词项之间的相关度如何？与一组查询项最相关的文档是哪些？原始的文档 – 词项矩阵为解决这些问题提供了一个简单的方法。我们可以通过计算矩阵中两个列向量之间的余弦相似度得到两个词项的相关度得分。**余弦相似度**度量了两个向量之间的夹角。在高维文档空间中，方向相同的两个向量彼此是相关的。两个向量的余弦相似度可以通过它们的点积除以向量的长度来得到。

自然语言处理和信息检索应用广泛采用余弦相似度作为度量词项和文档权重向量相似性的指标。类似地，两个文档的相关度得分可以通过这两个文档对应的两个行向量之间的余弦相似度来计算。词项和文档之间的相关度得分就更简单了，就是矩阵中相应行列的相交点。

但是，上述相似度计算是基于词项和文档相互关系的粗浅理解，依赖简单的频率计算。LSA 算法可以基于对语料库更深层次的理解来计算相似度得分。举个例子来说，即使词项 artillery 在文章“Normandy landings”中没有出现，LSA 算法也可以根据 artillery 和 howitzer 在其他文档中同时出现来发掘 artillery 和文章的关系。

LSA 算法的另一个优点就是效率高。它将重要信息表示为低维向量，这样我们就不用处理原始的文档 – 词项矩阵。考虑给定一个词项寻找与之最相关的其他词项的问题。如果采用前面提到的粗浅方法，我们需要计算词项列向量和文档 – 词项矩阵中所有其他列向量的点积。这里需要的乘法运行次数和词项个数与文档个数的乘积成正比。通过采用概念空间的表示并将其映射回词项空间，LSA 算法可以达到相同的效果，但乘法运算的次数与词项个数和 k 的乘积成正比。低阶近似对数据相关模式进行了编码，所以我们无须查询整个语料库。

在最后一节中，我们将使用 LSA 表示的数据来构建一个简单的查询引擎。本节的代码封装在本书代码库的 `LSAQueryEngine` 类中。

6.9 词项–词项相关度

LSA 将词项之间的关系解释为重构出来的低阶近似阵中的两个列之间的余弦相似度。也就是说，该矩阵可以通过将 3 个近似因子阵相乘得到。LSA 算法背后的思想是这个低阶矩阵

是对数据更有用的表示。这些用途表现在以下几个方面：

- 通过合并相关词项来处理同义词
- 通过对词项的不同含义赋予低的权重来处理多义词
- 过滤噪声

但是，要得到余弦相似度，不一定需要计算出矩阵的元素。线性代数运算告诉我们重构矩阵中的两个列的余弦相似度正好等于 SV^T 的相应列的余弦相似度。考虑寻找一个词项最相关的一组词项的问题。计算词项与其他所有词项之间的余弦相似度等价于先将 *VS* 中的每一行归一化，然后乘以词项对应的行。得到的结果向量中每个元素代表了词项与查询项之间的相似度。

为了节省篇幅，我们省略了计算 *VS* 和行归一化方法的实现代码，大家可以参考本书附带的 GitHub 资料库。我们在类 `LSAQuerayEngine` 初始化的过程中执行这些代码，以便日后可以重用：

```
import breeze.linalg.{DenseMatrix => BDenseMatrix}

class LSAQueryEngine(
    val svd: SingularValueDecomposition[RowMatrix, Matrix],
    ...
  ) {

  val VS: BDenseMatrix[Double] = multiplyByDiagonalMatrix(svd.V, svd.s)
  val normalizedVS: BDenseMatrix[Double] = rowsNormalized(VS)
  ...
```

在初始化的时候，计算 id-to-document 和 id-to-term 的逆向映射，这样我们就可以将查询到的字符串映射回矩阵中的位置：

```
val idTerms: Map[String, Int] = termIds.zipWithIndex.toMap
val idDocs: Map[String, Long] = docIds.map(_.swap)
```

现在，寻找与词项相关的词项：

```
def topTermsForTerm(termId: Int): Seq[(Double, Int)] = {
  val rowVec = normalizedVS(termId, ::).t ❶

  val termScores = (normalizedVS * termRowVec).toArray.zipWithIndex ❷

  termScores.sortBy(-_._1).take(10) ❸
}

def printTopTermsForTerm(term: String): Unit = {
  val idWeights = topTermsForTerm(idTerms(term))
  println(idWeights.map { case (score, id) =>
    (termIds(id), score) ❹
  }.mkString(", "))
}
```

❶ 在 *VS* 中查询给定词项 ID 对应的行。

❷ 计算每个词项的得分。

❸ 找出最高得分的词项。

❹ 计算词项到词项 ID 的映射。

如果你正在使用 `spark-shell`，就可以用以下方法加载此功能：

```
import com.cloudera.datascience.lsa.LSAQueryEngine

val termIdfs = idfModel.idf.toArray
val queryEngine = new LSAQueryEngine(svd, termIds, docIds, termIdfs)
```

下面是一些样例词项的最相关的词项得分情况：

```
queryEngine.printTopTermsForTerm("algorithm")

(algorithm,1.000000000000002), (heuristic,0.8773199836391916),
(compute,0.8561015487853708), (constraint,0.8370707630657652),
(optimization,0.8331940333186296), (complexity,0.823738607119692),
(algorithmic,0.8227315888559854), (iterative,0.822364922633442),
(recursive,0.8176921180556759), (minimization,0.8160188481409465)

queryEngine.printTopTermsForTerm("radiohead")

(radiohead,0.9999999999999993), (lyrically,0.8837403315233519),
(catchy,0.8780717902060333), (riff,0.861326571452104),
(lyricsthe,0.8460798060853993), (lyric,0.8434937575368959),
(upbeat,0.8410212279939793), (song,0.8280655506697948),
(musically,0.8239497926624353), (anthemic,0.8207874883055177)

queryEngine.printTopTermsForTerm("tarantino")

(tarantino,1.0), (soderbergh,0.780999345687437),
(buscemi,0.7386998898933894), (screenplay,0.7347041267543623),
(spielberg,0.7342534745182226), (dicaprio,0.7279146798149239),
(filmmaking,0.7261103750076819), (lumet,0.7259812377657624),
(directorial,0.7195131565316943), (biopic,0.7164037755577743)
```

6.10 文档–文档相关度

同样可以计算文档之间的相关度。要计算两个文档之间的相似度，只要计算 $u_1^{\mathrm{T}}S$ 和 $u_2^{\mathrm{T}}S$ 之间的余弦相似度，这里 u_i 是 U 中词项 i 对应的行。要计算一个文档和其他所有文档的相似度，只要计算 normalized(US) u_t。

与 normalized(VS) 类似，我们在 `LSAQueryEngine` 类中计算 normalized(US)，以便重用结果。由于 U 背后是一个 RDD，而不是本地矩阵，所以这里的实现方法稍有不同：

```
val US: RowMatrix = multiplyByDiagonalRowMatrix(svd.U, svd.s)
val normalizedUS: RowMatrix = distributedRowsNormalized(US)
```

然后，查找与文档相关的文档：

```
import org.apache.spark.mllib.linalg.Matrices

def topDocsForDoc(docId: Long): Seq[(Double, Long)] = {
  val docRowArr = normalizedUS.rows.zipWithUniqueId.map(_.swap)
    .lookup(docId).head.toArray ❶
  val docRowVec = Matrices.dense(docRowArr.length, 1, docRowArr)

  val docScores = normalizedUS.multiply(docRowVec) ❷

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId() ❸

  allDocWeights.filter(!_._1.isNaN).top(10) ❹
}

def printTopDocsForDoc(doc: String): Unit = {
  val idWeights = topDocsForDoc(idDocs(doc))
  println(idWeights.map { case (score, id) =>
    (docIds(id), score)
  }.mkString(", "))
}
```

❶ 在 *US* 中查找给定 doc ID 对应的行。

❷ 计算每个文档的得分。

❸ 找出得分最高的文档。

❹ 如果 *U* 中对应行元素全为 0，则文档得分可能是 NaN。所以需要将这些行去掉。

下面给出一些样例文档的最相似文档：

```
queryEngine.printTopDocsForDoc("Romania")

(Romania,0.9999999999999994), (Roma in Romania,0.9229332158078395),
(Kingdom of Romania,0.9176138537751187),
(Anti-Romanian discrimination,0.9131983116426412),
(Timeline of Romanian history,0.9124093989500675),
(Romania and the euro,0.9123191881625798),
(History of Romania,0.9095848558045102),
(Romania–United States relations,0.9016913779787574),
(Wiesel Commission,0.9016106300096606),
(List of Romania-related topics,0.8981305676612493)

queryEngine.printTopDocsForDoc("Brad Pitt")

(Brad Pitt,0.9999999999999984), (Aaron Eckhart,0.8935447577397551),
(Leonardo DiCaprio,0.8930359829082504), (Winona Ryder,0.8903497762653693),
(Ryan Phillippe,0.8847178312465214), (Claudette Colbert,0.8812403821804665),
(Clint Eastwood,0.8785765085978459), (Reese Witherspoon,0.876540742663427),
```

```
(Meryl Streep in the 2000s,0.8751593996242115),
(Kate Winslet,0.873124888198288)

queryEngine.printTopDocsForDoc("Radiohead")

(Radiohead,1.0000000000000016), (Fightstar,0.9461712602479349),
(R.E.M.,0.9456251852095919), (Incubus (band),0.9434650141836163),
(Audioslave,0.9411291455765148), (Tonic (band),0.9374518874425788),
(Depeche Mode,0.9370085419199352), (Megadeth,0.9355302294384438),
(Alice in Chains,0.9347862053793862), (Blur (band),0.9347436350811016)
```

6.11　文档–词项相关度

怎样计算词项和文档之间的相关度？这等价于计算文档 – 词项矩阵的低阶近似阵相应词项与文档对应的元素，即 $u_d^\mathrm{T}Sv_t$，其中 u_d 是 U 中文档对应的行，v_t 是 V 中词项对应的行。经过简单的线性代数运算就可以看出，词项和**每个**文档的相似度就是 $US\ v_t$。结果向量中的每个元素代表文档与查询项之间的相似度。另一个方向上文档与每个词项的相似度是 $u_d^\mathrm{T}SV$：

```
def topDocsForTerm(termId: Int): Seq[(Double, Long)] = {
  val rowArr = (0 until svd.V.numCols).
    map(i => svd.V(termId, i)).toArray
  val rowVec = Matrices.dense(termRowArr.length, 1, termRowArr)

  val docScores = US.multiply(rowVec) ❶

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId() ❷
  allDocWeights.top(10)
}

def printTopDocsForTerm(term: String): Unit = {
  val idWeights = topDocsForTerm(US, svd.V, idTerms(term))
  println(idWeights.map { case (score, id) =>
    (docIds(id), score)
  }.mkString(", "))
}
```

❶ 计算每个文档的得分。

❷ 找出得分最高的文档。

```
queryEngine.printTopDocsForTerm("fir")

(Silver tree,0.006292909647173194),
(See the forest for the trees,0.004785047583508223),
(Eucalyptus tree,0.004592837783089319),
(Sequoia tree,0.004497446632469554),
(Willow tree,0.004442871594515006),
(Coniferous tree,0.004429936059594164),
(Tulip Tree,0.004420469113273123),
```

```
(National tree,0.004381572286629475),
(Cottonwood tree,0.004374705020233878),
(Juniper Tree,0.004370895085141889)

queryEngine.printTopDocsForTerm("graph")

(K-factor (graph theory),0.07074443599385992),
(Mesh Graph,0.05843133228896666), (Mesh graph,0.05843133228896666),
(Grid Graph,0.05762071784234877), (Grid graph,0.05762071784234877),
(Graph factor,0.056799669054782564), (Graph (economics),0.05603848473056094),
(Skin graph,0.05512936759365371), (Edgeless graph,0.05507918292342141),
(Traversable graph,0.05507918292342141)
```

6.12 多词项查询

最后，我们该怎样实现多个词项的查询？找到与单个词项相关的文档只需从 V 中选出与该词项相应的行。这等价于用只有一个非零元素的词向量乘以 V。相反，如果是多个词项，只要用包含多个非零元素的词向量乘以 V，从而计算出概念 – 空间向量。为了保持原始文档 – 词项矩阵的权重机制，将查询中的词项权重值设为词项的逆文档频率。

```
termIdfs = idfModel.idf.toArray
```

从某种意义上说，这种查询方式就像是先在只有几个词项的语料库中增加文档，将该文档对应低阶近似文档 – 词项矩阵的一行，然后求该行与矩阵中其他行的余弦相似度。

```
import breeze.linalg.{SparseVector => BSparseVector}

def termsToQueryVector(terms: Seq[String])
  : BSparseVector[Double] = {
  val indices = terms.map(idTerms(_)).toArray
  val values = terms.map(idfs(_)).toArray
  new BSparseVector[Double](indices, values, idTerms.size)
}

def topDocsForTermQuery(query: BSparseVector[Double])
  : Seq[(Double, Long)] = {
  val breezeV = new BDenseMatrix[Double](V.numRows, V.numCols,
    V.toArray)
  val termRowArr = (breezeV.t * query).toArray

  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)

  val docScores = US.multiply(termRowVec) ❶

  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId() ❷
  allDocWeights.top(10)
}

def printTopDocsForTermQuery(terms: Seq[String]): Unit = {
```

```
    val queryVec = termsToQueryVector(terms)
    val idWeights = topDocsForTermQuery(queryVec)
    println(idWeights.map { case (score, id) =>
      (docIds(id), score)
    }.mkString(", "))
  }
```

❶ 计算每个文档的得分。

❷ 找出得分最高的文档。

```
queryEngine.printTopDocsForTermQuery(Seq("factorization", "decomposition"))

(K-factor (graph theory),0.04335677416674133),
(Matrix Algebra,0.038074479507460755),
(Matrix algebra,0.038074479507460755),
(Zero Theorem,0.03758005783639301),
(Birkhoff-von Neumann Theorem,0.03594539874814679),
(Enumeration theorem,0.03498444607374629),
(Pythagoras' theorem,0.03489110483887526),
(Thales theorem,0.03481592682203685),
(Cpt theorem,0.03478175099368145),
(Fuss' theorem,0.034739350150484904)
```

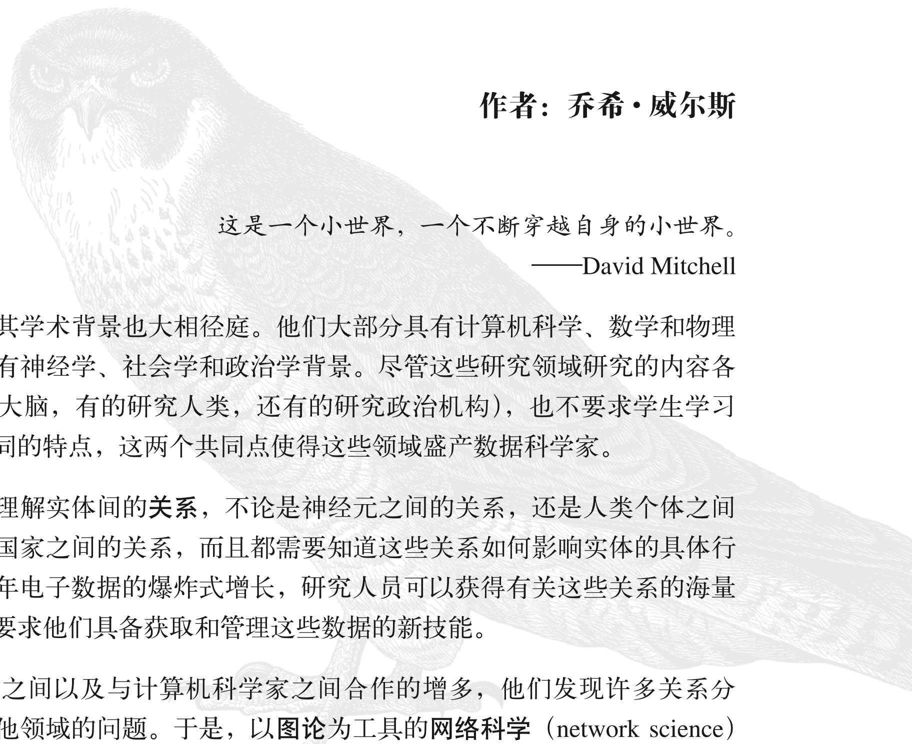
6.13 小结

除了用于文本分析，奇异值分解（SVD）和它的姊妹技术主成分分析（PCA）还有很多其他应用。**特征脸**（eigenface）是人脸识别的常用方法，它采用了这种技术来理解人类脸部的不同模式。在气象学研究中，可以使用这个技术从包含噪声的不同数据源中发现类似年轮一样的全球气温变化趋势。Michael Mann 著名的“曲棍球棒图”（https://en.wikipedia.org/wiki/Hockey_stick_controversy）描绘了整个 20 世纪气温升高的规律，这其实就是所谓的**概念**。SVD 和 PCA 算法也用于高维数据集的可视化。但一个数据集被归约为最重要的两到三个概念后，我们就可以将它绘制成人类可以观察的图形了。

还有许多其他方法可以用于理解海量文本语料库。比如潜在狄氏配置（LDA，latent Dirichlet allocation，https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation）就是其中一种技术。作为一个**话题模型**，该技术可以从语料库中推断出一组话题并计算出每个文档参与该话题的程度。

第7章

用GraphX分析伴生网络



作者：乔希·威尔斯

> 这是一个小世界，一个不断穿越自身的小世界。
>
> ——David Mitchell

数据科学家形形色色，其学术背景也大相径庭。他们大部分具有计算机科学、数学和物理学背景，但也有不少具有神经学、社会学和政治学背景。尽管这些研究领域研究的内容各不相同（比如有的研究大脑，有的研究人类，还有的研究政治机构），也不要求学生学习编程，但它们有两个共同的特点，这两个共同点使得这些领域盛产数据科学家。

第一，这些领域都需要理解实体间的**关系**，不论是神经元之间的关系，还是人类个体之间的关系，抑或是国家与国家之间的关系，而且都需要知道这些关系如何影响实体的具体行为。第二，随着过去十年电子数据的爆炸式增长，研究人员可以获得有关这些关系的海量信息，而这些海量数据要求他们具备获取和管理这些数据的新技能。

随着这些研究人员彼此之间以及与计算机科学家之间合作的增多，他们发现许多关系分析技术也能用于解决其他领域的问题。于是，以**图论**为工具的**网络科学**（network science）就诞生了。图论是一个数学学科，研究一组实体（称为**顶点**）之间两两关系（称为**边**）的特点。图论也被广泛应用于计算机科学，比如研究数据结构、计算机架构和网络设计（比如互联网等）。

图论和网络科学也对商业领域产生了深远影响。几乎所有大型互联网公司都建立了关系网

络，通过对这些重要的关系网络进行分析，这些公司获得了巨大的价值和更多的竞争优势。亚马逊和 Netflix 分别建立并掌握了顾客 – 商品购买关系和用户 – 电影评分关系，并且基于这些关系构建各自的推荐算法。Facebook 和 LinkedIn 则构建了人类关系图谱并对这些关系进行分析，目的就是为了更好地组织内容、提升广告效果以及帮助人们建立新的关系。在构建关系网络方面，最有名的例子可能要数谷歌创始人发明的 PageRank 算法，该算法从根本上改善了互联网的搜索方式。

这些以网络为中心的公司的计算和分析需求促使了 MapReduce 等分布式处理框架的产生。与此同时，这些公司也雇用了许多数据科学家，使用这些工具对越来越多的数据进行更快的分析。MapReduce 框架最初就是为了求解 PageRank 算法核心方程式设计的，该框架提供了一种可扩展和可靠的方式。随着图谱越来越大，数据科学家需要更快地进行分析，于是不断有新的并行图处理框架被开发出来，比如谷歌的 Pregel、雅虎的 Giraph 和卡内基梅隆大学的 GraphLab。这些框架以图处理为中心，支持容错和迭代式内存计算，对某些类型的图计算的处理效率大大高于对应的数据并行式 MapReduce 作业。

本章将介绍 Spark 之上的一个扩展工具 GraphX，它支持 Pregel、Giraph 和 GraphLab 中的许多图并行处理任务。相比于 Pregel、Giraph 和 GraphLab 这些定制的图计算框架，GraphX 无法在每个图计算上都与它们一样快，但由于 GraphX 基于 Spark，在进行数据分析过程中，如果你想引入 GraphX 对以网络为中心的数据集进行分析是非常方便的。有了 GraphX，你能够像往常一样使用熟悉的 Spark 抽象来进行图并行编程。

GraphX 与 GraphFrame

GraphX 在 Spark 1.3 版本引入 DataFrame 之前就存在，它的 API 都是基于 RDD 而设计的。最近社区正在努力将 GraphX 移植到基于 DataFrame 的新 API 上，因此它被贴切地称为 GraphFrame。与传统的 GraphX API 相比，GraphFrame 有许多优点，包括通过 DataFrame API 对图格式进行序列化和反序列化的更加丰富的功能支撑，以及表达力更强的图查询。

在编写本书第 2 版的时候，Spark 2.1 的 GraphFrame 并不能处理本章中的所有分析，所以我们决定继续使用功能完整的 GraphX API。好消息是，GraphFrame API 的很多理念来自 GraphX，因此本章中介绍的所有方法和概念都能和 GraphFrame 一一对应起来。我们期待着在本书的下一个版本（目前纯粹是构想）中迁移到 GraphFrame；你也可以跟踪 Graph Frame 项目的进展状态（http://graphframes.github.io/）。

7.1 对MEDLINE文献引用索引的网络分析

MEDLINE（Medical Literature Analysis and Retrieval System Online，医学文献在线分析和检索系统）是一个学术论文数据库，收录发表在生命科学和医学领域期刊上的文献。MEDLINE 由美国国家卫生研究院（National Institute of Health，NIH）下属的国家医学图

书馆（National Library of Medicine，NLM）管理并发行。它的文献引用索引记录了数千种期刊上发表的论文，最早的论文可以追溯到 1879 年。从 1971 年开始 MEDLINE 对医科学校提供在线访问，从 1996 年开始对外提供公开访问的网页。主数据库收录了 2000 多万篇文章，其中最早的文章发表在 20 世纪 50 年代初期，该数据库每个工作日都更新。

由于 MEDLINE 引用量非常大而且更新频率快，研究人员在所有文献引用索引上开发了一套全面的语义标签，称为 MeSH（Medical Subject Headings）。这些标签提供了一个有用的框架，使用 MeSH，人们就可以在阅读文献时知道文献之间的关系。同时人们基于 MeSH 开发了许多数据产品：2001 年 PubGene 向人们展示了第一个生物医学文本挖掘的生产应用。这是一个搜索引擎，人们可以利用它查看 MeSH 术语的关系图谱，而正是这个图谱将文档连接在一起。

本章我们将使用 Scala、Spark 和 GraphX 来获取、转换并分析 MeSH 术语网络，这些术语来自 MEDLINE 近期公开的引用数据子集。我们的网络分析思想来自于 Kastrin 等于 2014 年发表的论文“Large-Scale Structure of a Network of Co-Occurring MeSH Terms: Statistical Analysis of Macroscopic Properties”（https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4090190/）。但我们使用了一个不同的引用数据子集，同时原论文中采用的是 R 工具包和 C++ 代码，我们这里使用 GraphX。

我们的目的是了解文献引用图谱的概况和特点。为了实现这个目标，我们要从多个角度来研究数据集，这样才能全面了解数据集。首先，我们研究数据集中的主要主题和它们的伴生关系，这个分析比较简单，不需要使用 GraphX。然后，我们要找出数据集中的**连通组件**（connected component）。我们能不能沿着一条引用路径从一个主题达到另一个主题？数据实际上是不是由一系列独立的子图组成的？这些都是我们要回答的问题。接下来，我们会继续讨论**图的度分布**，它描述了主题的相关度变化并有助于我们找到那些与其他主题关联最多的主题。最后，我们要计算两个图统计量：**聚类系数**和**平均路径长度**，它们的复杂度稍微高一点儿。这些统计量有助于我们了解文献引用图谱与万维网和 Facebook 社交网络等实际图谱的相似程度。

7.2 获取数据

我们可以从 NIH 的 FTP 服务器上下载文献引用数据的一个样本，脚本如下：

```
$ mkdir medline_data
$ cd medline_data
$ wget ftp://ftp.nlm.nih.gov/nlmdata/sample/medline/*.gz
```

解压并检查数据，然后将数据上传到 HDFS 上，代码如下：

```
$ gunzip *.gz
$ ls -ltr
```

```
...
total 1814128
-rw-r--r--  1 jwills  staff  145188012 Dec  3  2015 medsamp2016h.xml
-rw-r--r--  1 jwills  staff  133663105 Dec  3  2015 medsamp2016g.xml
-rw-r--r--  1 jwills  staff  131298588 Dec  3  2015 medsamp2016f.xml
-rw-r--r--  1 jwills  staff  156910066 Dec  3  2015 medsamp2016e.xml
-rw-r--r--  1 jwills  staff  112711106 Dec  3  2015 medsamp2016d.xml
-rw-r--r--  1 jwills  staff  105189622 Dec  3  2015 medsamp2016c.xml
-rw-r--r--  1 jwills  staff   72705330 Dec  3  2015 medsamp2016b.xml
-rw-r--r--  1 jwills  staff   71147066 Dec  3  2015 medsamp2016a.xml
```

样本数据格式为 XML，解压之后大约有 600 MB。样本文件中每个条目是一条 `MedlineCitation` 类型的记录，该记录包含文章在生物医学杂志上的发表信息，包括杂志名称、发行期号、发行日期、作者姓名、摘要、MeSH 关键字集合。此外，MeSH 关键字还有一个属性，用于表示该关键字所指概念是不是文章主要的主题。我们来看看 medsamp2016a.xml 文件中第一个引用记录：

```
<MedlineCitation Owner="PIP" Status="MEDLINE">
<PMID Version="1">12255379</PMID>
<DateCreated>
  <Year>1980</Year>
  <Month>01</Month>
  <Day>03</Day>
</DateCreated>
...
<MeshHeadingList>
...
  <MeshHeading>
    <DescriptorName MajorTopicYN="N">Humans</DescriptorName>
  </MeshHeading>
  <MeshHeading>
    <DescriptorName MajorTopicYN="Y">Intelligence</DescriptorName>
  </MeshHeading>
  <MeshHeading>
    <DescriptorName MajorTopicYN="Y">Rorschach Test</DescriptorName>
  </MeshHeading>
...
</MeshHeadingList>
...
</MedlineCitation>
```

前一章在对维基百科文章进行潜在语义分析时，我们主要关心 XML 记录中的非结构化文本。但对本章中伴生关系的分析，我们直接通过解析 XML 结构并从 `DescriptorName` 标签中提取值。幸运的是，Scala 提供了一个非常优秀的工具 scala-xml，可以直接用它来解析和查询 XML 文档。

先将引用数据加载到 HDFS 上：

```
$ hadoop fs -mkdir medline
$ hadoop fs -put *.xml medline
```

现在启动 Spark shell。本章要用到第 6 章解析 XML 格式数据的代码。为了将这些代码编译成一个可供引用的 JAR，需要先到 Git 资料库上的 ch07-graph/ 目录下并执行 Maven 构建：

```
$ cd ch07-graph/
$ mvn package
$ cd target
$ spark-shell --jars ch07-graph-2.0.0-jar-with-dependencies.jar
```

为了把 XML 格式的 MEDLINE 数据读到 Spark shell 中，我们需要实现一个函数，代码如下：

```
import edu.umd.cloud9.collection.XMLInputFormat
import org.apache.spark.sql.{Dataset, SparkSession}
import org.apache.hadoop.io.{Text, LongWritable}
import org.apache.hadoop.conf.Configuration

def loadMedline(spark: SparkSession, path: String) = {
  import spark.implicits._
  @transient val conf = new Configuration()
  conf.set(XMLInputFormat.START_TAG_KEY, "<MedlineCitation ")
  conf.set(XMLInputFormat.END_TAG_KEY, "</MedlineCitation>")
  val sc = spark.sparkContext
  val in = sc.newAPIHadoopFile(path, classOf[XMLInputFormat],
      classOf[LongWritable], classOf[Text], conf)
  in.map(line => line._2.toString).toDS()
}
val medlineRaw = loadMedline(spark, "medline")
```

由于不同记录中 `MedlineCitation` 开始标签中属性值可能不同，所以我们将配置参数 `START_TAG_KEY` 的值设为 `MedlineCitation` 开始标签的前缀。`XmlInputFormat` 会在返回的记录值中包含这些不同的属性。

7.3　用Scala XML工具解析XML文档

Scala 和 XML 之间有一段渊源比较有意思。从 1.2 版本开始，Scala 就把 XML 作为它的一级数据类型，所以下面这段代码的句法是没问题的：

```
import scala.xml._

val cit = <MedlineCitation>data</MedlineCitation>
```

这种对 XML 字面量的支持在主流编程语言中不多见，特别是像 JSON 这类序列化格式被广泛采用之后。对此，Martin Odersky 在 2012 年 Scala 语言邮件列表中做出如下评论：

> [XML 字面量] 在当时是个不错的特性，但现在看起来它有点儿不合时宜。相信有了新的字符串插入方案后，就可以把所有 XML 处理代码都放到库里了，这应该是个不小的进步。

从 Scala 2.11 开始，`scala.xml` 包不再包含在 Scala 核心库之中。在将 Scala 升级到 Scala

2.11 之后，如果需要在项目中使用 Scala XML 工具包，我们必须显式地将 `scala-xml` 依赖包包含进来。

记住上述要点之后，我们必须承认 Scala 对 XML 文档的解析和查询真的非常优秀，从 MEDLINE 引用数据中提取信息时我们要反复利用这种优点。现在就开始把第一条未解析的引用记录提取到我们的 Spark shell 中吧：

```
val rawXml = medlineRaw.take(1)(0)
val elem = XML.loadString(rawXml)
```

变量 `elem` 是 `scala.xml.Elem` 类的实例，Scala 用 `scala.xml.Elem` 类表示 XML 文档中的一个节点，该类内置了查询节点信息和节点内容的函数，比如：

```
elem.label
elem.attributes
```

它也提供了查找给定 XML 节点子节点的几个运算符，其中第一个就是 `\`，用于根据名称查询节点的直接子节点：

```
elem \ "MeshHeadingList"
...
NodeSeq(<MeshHeadingList>
<MeshHeading>
<DescriptorName MajorTopicYN="N">Behavior</DescriptorName>
</MeshHeading>
...
```

运算符 `\` 只对节点的**直接**子节点有效，如果我们运行 `elem \ "MeshHeading"`，结果会是一个空的 `NodeSeq`。为了得到给定节点的间接子节点，我们要用运算符 `\\`：

```
elem \\ "MeshHeading"
...
NodeSeq(<MeshHeading>
<DescriptorName MajorTopicYN="N">Behavior</DescriptorName>
</MeshHeading>,
...
```

可以用 `\\` 运算符直接得到 `DescriptorName` 条目，并通过在每个 `NodeSeq` 内部元素上调用 `text` 函数把每个节点内的 MeSH 标签提取出来：

```
(elem \\ "DescriptorName").map(_.text)
...
List(Behavior, Congenital Abnormalities, ...
```

最后请注意，每个 `DescriptorName` 条目都有一个 `MajorTopicYN` 属性，它表示该 MeSH 标签是否是所引用的文章的主要主题。只要我们在 XML 标签属性前加上“@”符号，就可以用 `\` 和 `\\` 运算符得到 XML 标签属性的值。用这个特性我们可以建立一个过滤器，只返

回文章的主要 MeSH 标签的名称，代码如下：

```
def majorTopics(record: String): Seq[String] = {
  val elem = XML.loadString(record)
  val dn = elem \\ "DescriptorName"
  val mt = dn.filter(n => (n \ "@MajorTopicYN").text == "Y")
  mt.map(n => n.text)
}
majorTopics(elem)
```

现在我们的 XML 解析代码可以在本地运行了，我们可以用它解析 RDD 中每条引用记录的 MeSH 编码并将结果缓存起来：

```
val medline = medlineRaw.map(majorTopics)
medline.cache()
medline.take(1)(0)
```

7.4　分析MeSH主要主题及其伴生关系

在将 MEDLINE 引用记录中所需的 MeSH 标签提取出来之后，我们需要知道数据集中标签的总体分布情况。为此我们需要使用 Spark SQL 计算一些基本统计量，比如记录条数和主要 MeSH 主题出现频率的直方图，代码如下：

```
medline.count()
val topics = medline.flatMap(mesh => mesh).toDF("topic")
topics.createOrReplaceTempView("topics")
val topicDist = spark.sql("""
  SELECT topic, COUNT(*) cnt
  FROM topics
  GROUP BY topic
  ORDER BY cnt DESC""")
topicDist.count()
...
res: Long = 14548
topicDist.show()
...
+--------------------+----+
|               topic| cnt|
+--------------------+----+
|            Research|1649|
|             Disease|1349|
|           Neoplasms|1123|
|        Tuberculosis|1066|
|       Public Policy| 816|
|       Jurisprudence| 796|
|          Demography| 763|
| Population Dynamics| 753|
|           Economics| 690|
|            Medicine| 682|
...
```

出现最频繁的主要主题是那些最常见的主题，比如超级常见的 Research 和 Disease，以及稍微弱一点儿的 Neoplasms 和 Tuberculosis，这一点在我们的意料之中。好在我们的数据集中有超过 13 000 个不同的主要主题，考虑到最常出现的主题只出现了少数（1649/240 000，约为 0.7%）文档中，我们估计包含某个主题的文档的个数的总体分布呈现长尾形态。为了验证我们的估计，我们对 `topicDist` DataFrame 中出现次数相同的主题的个数进行统计：

```
topicDist.createOrReplaceTempView("topic_dist")
spark.sql("""
  SELECT cnt, COUNT(*) dist
  FROM topic_dist
  GROUP BY cnt
  ORDER BY dist DESC
  LIMIT 10""").show()
...
+---+----+
|cnt|dist|
+---+----+
|  1|3106|
|  2|1699|
|  3|1207|
|  4| 902|
|  5| 680|
|  6| 571|
|  7| 490|
|  8| 380|
|  9| 356|
| 10| 296|
+---+----+
```

当然我们主要还是关心 MeSH 的伴生主题。MEDLINE 数据集中每一项都是一个字符串列表，代表每个引用记录中提及的主题名称。要得到伴生主题，我们要为这些字符串列表生成一个二元组集合。幸运的是，Scala 的集合工具包有一个 `combinations` 方法，利用这个方法产生这些子列表极其容易。`combinations` 方法返回的是一个 `Iterator`，因此并不需要同时把所有组合都放在内存里。

```
val list = List(1, 2, 3)
val combs = list.combinations(2)
combs.foreach(println)
```

当用这个函数来生成子列表时，我们要注意所有的列表要按照同样的方式进行排序，以便之后使用 Spark 对它们进行汇总。这是因为 `combinations` 函数返回的列表取决于输入元素的顺序，而元素相同但顺序不同的两个列表是不相等的。

```
val combs = list.reverse.combinations(2)
combs.foreach(println)
List(3, 2) == List(2, 3)
```

因此，如果我们要为每条引用记录生成二元子列表集合，在调用 `combinations` 之前要确保

主题列表是排好序的：

```
val topicPairs = medline.flatMap(t => {
  t.sorted.combinations(2)
}).toDF("pairs")
topicPairs.createOrReplaceTempView("topic_pairs")
val cooccurs =  spark.sql("""
  SELECT pairs, COUNT(*) cnt
  FROM topic_pairs
  GROUP BY pairs""")
cooccurs.cache()
cooccurs.count()
```

由于我们的数据中有 14 548 个主题，总共可能有 14 548*14 547/2 = 105 814 878 个无序的伴生二元组。然而，伴生组的计数结果显示数据集中实际上只有 213 745 组，只占可能数量的很小一部分。如果考察一下数据中最常出现的伴生二元组，我们可以得到如下结果：

```
cooccurs.createOrReplaceTempView("cooccurs")
spark.sql("""
  SELECT pairs, cnt
  FROM cooccurs
  ORDER BY cnt DESC
  LIMIT 10""").collect().foreach(println)
...
[WrappedArray(Demography, Population Dynamics),288]
[WrappedArray(Government Regulation, Social Control, Formal),254]
[WrappedArray(Emigration and Immigration, Population Dynamics),230]
[WrappedArray(Acquired Immunodeficiency Syndrome, HIV Infections),220]
[WrappedArray(Antibiotics, Antitubercular, Dermatologic Agents),205]
[WrappedArray(Analgesia, Anesthesia),183]
[WrappedArray(Economics, Population Dynamics),181]
[WrappedArray(Analgesia, Anesthesia and Analgesia),179]
[WrappedArray(Anesthesia, Anesthesia and Analgesia),177]
[WrappedArray(Population Dynamics, Population Growth),174]
```

最常出现的伴生二元组也没有什么特别之处，如果我们根据最常出现的主要主题来猜测，结果也差不多。出现最多的伴生二元组，例如 (Demography, Population Dynamics)（人口统计学，人口动力学）这个组合的出现。要么由于它们是两个独立主题，但都是最常出现的；要么由于它们频繁共现，基本上可以当成同义词，就像 (Analgesia, Anesthesia)（镇痛，麻醉）这种组合一样。这一点不足为奇，除了说明数据中存在伴生二元组之外也没有提供什么额外信息。

7.5 用GraphX来建立一个伴生网络

正如在前一节中所看到的那样，在研究伴生网络时标准的数据统计工具不能提供额外有价值的信息。我们能计算的总体概要统计量，比如原始记录条数等，无法让我们了解网络中关系的总体结构。并且运用这些工具得到的处于分布两端的伴生二元组常常是我们不太感兴趣的。

我们真正想要做的是分析伴生网络：把主题当作图的顶点，把连接两个主题的引用记录看成两个相应顶点之间的边。这样我们就可以计算以网络为中心的统计量。这些网络统计量能帮助我们理解网络的总体结构并识别出那些有意思的局部离群顶点，识别出这些离群点之后我们才需要对其进行进一步研究。

我们也可以利用伴生网络找出需要进一步研究的那些有意思的相互关系。图 7-1 是抗癌药物和病人服用药物所产生的不良反应的部分伴生关系图。我们可以利用从这些图中得到的信息设计临床实验并研究药物和不良反应的相互关系。

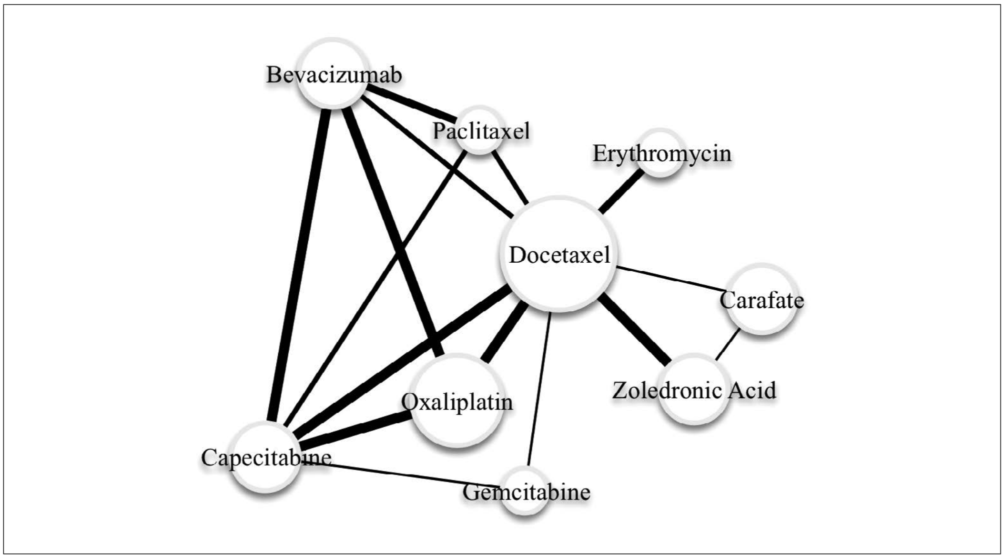


图 7-1：抗癌药物和相关病人不良反应组的部分伴生关系图

MLlib 对利用 Spark 建立机器学习模型提供了一组模式和算法。与之类似，设计 GraphX 这个 Spark 工具包是为了帮助我们利用图论的语言和工具分析各种网络。由于 GraphX 构建在 Spark 之上，它继承了 Spark 在可扩展性方面的所有特性。这就意味着可以利用 GraphX 对规模极其庞大的图进行分析，这些分析任务可以在多个机器上分布式执行。GraphX 也可以与 Spark 平台上的其他组件很好地集成在一起，这样数据科学家就能在各种任务中轻松切换：从编写运行在 RDD 之上的 ETL 任务切换到执行那些运行在图上的图并行算法，然后再切回到以数据并行的方式对图计算输出结果并进行分析和汇总。这一点我们将在本章中看到。GraphX 允许我们在分析过程中轻松引入图计算方式，正是因为这种无缝集成才使得 GraphX 变得如此强大。

像 Dataset API 一样，GraphX 也构建在 Spark 的基础数据单元 RDD 上。具体而言，GraphX 针对图计算对 RDD 的实现进行了两项特殊的优化。`VertexRDD[VD]` 是 `RDD[(VertexId, VD)]` 的特殊实现，其中 `VertexID` 类型是 `Long` 的实例，对每个顶点都是必需的。`VD` 是顶点关

联的任何类型数据，称为**顶点属性**（vertex attribute）。EdgeRDD[ED] 是 RDD[Edge[ED]] 的特殊实现，其中 Edge 是包含两个 VertexId 值和一个 ED 类型的**边属性**（edge attribute）。VertexRDD 和 EdgeRDD 在每个数据分区内部均有用于加快连接和属性更新的索引结构。给定 VertexRDD 及其相应的 EdgeRDD 后，我们就能建立一个 Graph 类的实例，Graph 类包含了许多图计算的高效方法。

要建立一个图，首先要取得用作图顶点标识符的 Long 型值。由于所有主题都是用字符串标示的，因此我们在创建伴生网络时要处理这个小问题。我们需要有一种方式将每个主题字符串转换为 64 位的 Long 型值，而且这种方式最好能够分布式执行。这样就能对全部数据进行快速处理。

方法之一就是用内置的 hashCode 方法来对任意 Scala 对象产生一个 32 位整数。就本章要讨论的问题而言，我们的图只有 13 000 个顶点，这种方法可能行得通。但对于一个有数百万甚至是数千万顶点的图来说，发生散列冲突的概率可能就太高了。因此我们从谷歌的 Guava 库复制散列实现。利用该工具，可以通过 MD5 散列算法为每个主题生成一个 64 位的唯一标识符，代码如下：

```
import java.nio.charset.StandardCharsets
import java.security.MessageDigest

def hashId(str: String): Long = {
 val bytes = MessageDigest.getInstance("MD5").
   digest(str.getBytes(StandardCharsets.UTF_8))
  (bytes(0) & 0xFFL) |
  ((bytes(1) & 0xFFL) << 8) |
  ((bytes(2) & 0xFFL) << 16) |
  ((bytes(3) & 0xFFL) << 24) |
  ((bytes(4) & 0xFFL) << 32) |
  ((bytes(5) & 0xFFL) << 40) |
  ((bytes(6) & 0xFFL) << 48) |
  ((bytes(7) & 0xFFL) << 56)
}
```

在 MEDLINE 数据上运行该散列函数可以得到一个 DataFrame，以它为基础就可以得到伴生关系图的顶点集合。我们还需要对结果进行简单的校验以确保每个主题的散列标识符是唯一的，代码如下：

```
import org.apache.spark.sql.Row
val vertices = topics.map{ case Row(topic: String) =>
(hashId(topic), topic) }.toDF("hash", "topic")
val uniqueHashes = vertices.agg(countDistinct("hash")).take(1)
...
res: Array[Row] = Array([14548])
```

我们要用前面一节中得到的伴生频率计数来生成图的边，方法是使用散列函数将每个主题的名称映射到相应的顶点 ID。在生成边的时候，一个好习惯就是要保证左边的 VertexId

（GraphX 称为 src）要比右边的 VertexId（GraphX 称为 dst）小。虽然 GraphX 工具包中大多数算法都不要求 src 和 dst 之间有大小关系，但确实有几个算法存在这样的要求。因此最好在一开始时就保证大小顺序，这样的话就再也不用为此操心了。

```
import org.apache.spark.graphx._

val edges = cooccurs.map{ case Row(topics: Seq[_], cnt: Long) =>
  val ids = topics.map(_.toString).map(hashId).sorted
  Edge(ids(0), ids(1), cnt)
}
```

把顶点和边都准备好之后就可以创建 Graph 实例了。我们需要将 Graph 缓存起来，这样便于后续处理时使用，代码如下：

```
val vertexRDD = vertices.rdd.map{
  case Row(hash: Long, topic: String) => (hash, topic)
}
val topicGraph = Graph(vertexRDD, edges.rdd)
topicGraph.cache()
```

用于创建 Graph 实例的 vertexRDD 和 edges 参数是普通的 RDD。我们甚至没有为保证每个主题实例的唯一性而对 vertices 进行去重。幸运的是，Graph API 帮我们完成了这项工作，它会将我们输入的 RDD 转换成一个 VertexRDD 和一个 EdgeRDD，这样顶点计数就是唯一的了：

```
vertexRDD.count()
...
280464

topicGraph.vertices.count()
...
14548
```

注意，如果某两个顶点二元组在 EdgeRDD 中重复出现，Graph API 不会对其进行去重处理，这样 GraphX 就可以创建**多图**（multigraph），也就是相同的顶点之间可以用多条不同值的边。如果图顶点代表了有许多丰富涵义的对象，多图往往是很有用的。比如人或公司，他们之间就可能有许多不同的关系（比如朋友、家庭成员、顾客、合作伙伴等）。多图也可以让我们根据实际情况使用有向边或无向边。

7.6　理解网络结构

研究表的内容时我们可以快速算出列上的很多概要统计量，这样就能大概知道数据的结构并可以对问题域作进一步研究。研究图时，这个原则同样适用，只不过此时我们感兴趣的概要统计量稍有不同。Graph 类内置了计算这些概要统计量的方法，结合其他常规的 Spark RDD API，我们可以很轻松地了解到图的结构，这样就能为研究图提供方向。

7.6.1 连通组件

对于图来说，我们想了解的一个基本情况就是它是否是**连通图**。对于连通图中的任意两个顶点，它们之间都存在一条**路径**到达对方，路径就是连接两个顶点的一系列边。如果图是非连通的，那么我们可以将图划分成一组更小的子图，这样就可以分别对每个子图进行研究。

连通性是图的基本属性，所以很自然地 GraphX 内置了找出图的连通组件的方法。如果在图上调用过 `connectedComponents` 方法，你就会注意到 Spark 会生成许多作业，等作业结束后，就会得到计算的结果：

```
val connectedComponentGraph = topicGraph.connectedComponents()
```

请注意 `connectedComponents` 方法返回对象的类型，也是 `Graph` 类型实例，但顶点属性的类型是 `VertexId`，它是每个顶点所属连通组件的唯一标识符。想得到连通组件的个数和大小，我们可以将 `VertexRDD` 转换回 DataFrame，然后使用我们的标准工具包：

```
val componentDF = connectedComponentGraph.vertices.toDF("vid", "cid")
val componentCounts = componentDF.groupBy("cid").count()
componentCounts.count()
...
878
```

我们先来看几个最大的连通组件：

```
componentCounts.orderBy(desc("count")).show()
...
+--------------------+-----+
|                 cid|count|
+--------------------+-----+
|-9218306090261648869|13610|
|-8193948242717911820|    5|
|-2062883918534425492|    4|
|-8679136035911620397|    3|
| 1765411469112156596|    3|
|-7016546051037489808|    3|
|-7685954109876710390|    3|
| -784187332742198415|    3|
| 2742772755763603550|    3|
...
```

最大的连通组件包含了超过 90% 的顶点，第二大的连通组件包含 4 个顶点，只占图的非常小的一部分。为了搞清楚为什么这些小组件没有和最大的组件连通，需要看一下它们的主题。为了查看小组件相关的主题的名称，需要将连通组件图对应的 `VertexRDD` 和原始概念图执行 `join` 操作。`VertexRDD` 提供了 `innerJoin` 转换，它利用了 GraphX 的内部数据结构，性能比常规 Spark 的 `join` 转换要好得多。`innerJoin` 方法需要我们提供一个函数，该函数的输入为 `VertexID` 和两个 `VertexRDD` 的内部数据，函数的返回值是一个新的 `VertexRDD`，它是 `innerJoin` 方法结果。对应到我们这里的情况，我们想要知道每个连通组件的概念的

名称，因此需要返回一个包含主题名称和组件 ID 的 DataFrame：

```
val topicComponentDF = topicGraph.vertices.innerJoin(
  connectedComponentGraph.vertices) {
  (topicId, name, componentId) => (name, componentId.toLong)
}.toDF("topic", "cid")
```

我们来看一下第二大连通组件的主题名称：

```
topicComponentDF.where("cid = -2062883918534425492").show()
...
+--------------------+--------------------+
|               topic|                 cid|
+--------------------+--------------------+
|          Serotyping|-2062883918534425492|
|Campylobacter jejuni|-2062883918534425492|
|Campylobacter Inf...|-2062883918534425492|
|  Campylobacter coli|-2062883918534425492|
+--------------------+--------------------+
```

在搜索引擎上查询一下，就能知道 Campylobacter 是一种细菌，是引起食物中毒最常见的病因之一，而 serotyping 是一种基于细胞表面“抗原”对细菌分类的技术，“抗原”是一种在体内引起免疫反应的毒素。（这种调研工作平均每天要数据科学家花掉至少两个小时在维基百科上。）

下面是最初的主题分布，看看数据集中是否有任何类似命名的主题，在该簇群中并未出现：

```
val campy = spark.sql("""
  SELECT *
  FROM topic_dist
  WHERE topic LIKE '%ampylobacter%'""")
campy.show()
...
+--------------------+---+
|               topic|cnt|
+--------------------+---+
|Campylobacter jejuni|  3|
|Campylobacter Inf...|  2|
|  Campylobacter coli|  1|
|       Campylobacter|  1|
| Campylobacter fetus|  1|
+--------------------+---+
```

Campylobacter fetus 这个主题听起来和我们的 Campylobacter 菌群相似，但是在 MEDLINE 引用数据中，没有一篇文章将二者关联起来。互联网上的进一步搜索结果显示，Campylobacter 的这个亚种主要在牛羊身上发现，与人类无关，因此尽管名称相似，在研究文献中却无法关联起来。

这里我们学到的一个新知识，随着我们不断加入论文引用数据，主题伴生网络就非常可能

变成全连通图，并且没有什么重要理由让我们相信这个伴生网络会分解成独立的子图。

底层实现上，为了找出每个顶点所属的连通组件，`connectedComponents` 方法利用 `VertexId` 作为顶点唯一标识符在图上执行一些列式迭代计算。在计算的每个阶段，每个顶点把它所收到的最小 `VertexID` 广播到相邻节点。第一次迭代时，这个最小 `VertexID` 就是顶点自身的 ID，在随后的迭代中该最小 `VertexID` 通常会被更新掉。每个顶点都记录它所收到的 `VertexID` 的最小值，如果在某一次迭代中，所有顶点的最小 `VertexID` 都没有变化，那么连通组件的计算就完成了，每个顶点都将分配给该顶点的最小 `VertexID` 所代表的组件。在图上的这种迭代式计算非常普遍，本章后面将介绍怎样使用这种迭代模式来计算其他图结构指标。

7.6.2　度的分布

连通图的结构可能有很多种。比如，它可能是一个节点和所有其他节点相连，而其他节点之间则不直接相连。如果我们去掉这个中心节点，图就散落成若干分离的顶点。图的结构也可能是每个顶点都只和两个其他顶点相连，整个组件形成一个巨大的环。

图 7-2 说明，即使同样的连通图，它们的度分布也可能迥异。

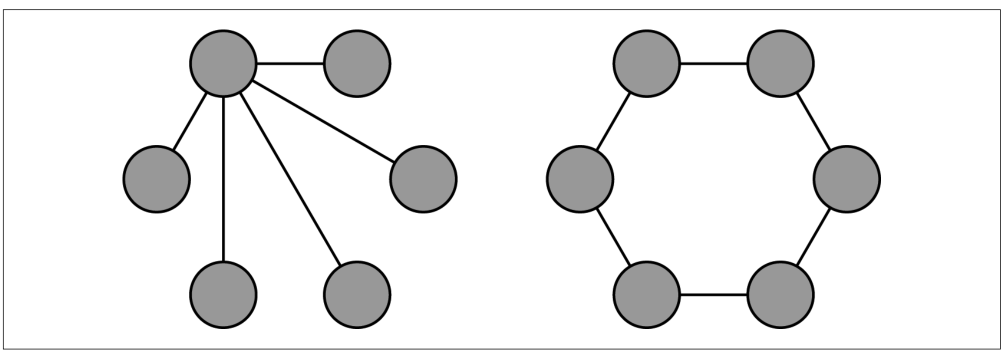


图 7-2：连通图的度分布

为了更多了解图的结构信息，我们需要知道每个顶点的度，也就是每个顶点所属边的条数。对于一个无环图（如果边的两个顶点相同就形成环），因为每条边都包含两个不同的顶点，所以全体顶点的度之和等于边的条数的两倍。

GraphX 中我们可以通过在 `Graph` 对象上调用 `degrees` 方法得到每个顶点的度。`degrees` 方法返回一个整数的 `VertexRDD`，其中每个整数代表一个顶点的度。现在我们来为概念网络计算度的分布和一些基本的概要统计量，代码如下：

```
val degrees: VertexRDD[Int] = topicGraph.degrees.cache()
degrees.map(_._2).stats()
...
```

```
(count: 13721, mean: 31.155892,
 stdev: 65.497591, max: 2596.000000,
 min: 1.000000)
```

度分布中有几个点要注意。第一，degrees RDD 中条目个数比概念图的顶点数少。概念图有 14 548 个顶点，而 degrees RDD 只有 13 721 个条目。有些顶点没有连接边。这可能是由于 MEDLINE 数据中某些引用只有一个主要主题词，因此有些主题并不和其他任何主题同时出现。可以通过重新检查原始数据集 medline 来确认我们的推测，代码如下：

```
val sing = medline.filter(x => x.size == 1)
sing.count()
...
44509

val singTopic = sing.flatMap(topic => topic).distinct()
singTopic.count()
...
8243
```

有 8243 个不同主题词单独出现在 MEDLINE 数据库中的 44 509 篇文章中。现在我们将已经在 topicPairs 数据集出现过的这些主题词去掉，代码如下：

```
val topic2 = topicPairs.flatMap(_.getAs[Seq[String]](0))
singTopic.except(topic2).count()
...
827
```

这会过滤掉 MEDLINE 数据库文档中单独出现的 827 个主题词。14 548 减去 827 等于 13 721，正好是 degrees RDD 中的条目数。

其次，请注意虽然度的均值较小，意味着普通顶点只连接到少数几个其他节点，但是度的最大值却表明至少有一个节点是高度连接的，它几乎和图中三分之一的顶点都是连接的。

我们来进一步看一下那些度很高的顶点所对应的概念，具体方法是使用 GraphX 的 innerJoin 在 degrees VertexRDD 和概念图中的顶点上执行 join 运算。这里我们还要提供一个关联函数将概念名称和顶点的度组织成一个二元组，请记住 innerJoin 方法只返回在两个 VertexRDD 中均出现的顶点，因此那些没有与其他概念同时出现的概念将被过滤掉。

```
val namesAndDegrees = degrees.innerJoin(topicGraph.vertices) {
  (topicId, degree, name) => (name, degree.toInt)
}.values.toDF("topic", "degree")
```

如果我们打印出 namesAndDegrees DataFrame 中度数最高的 10 个顶点，会得到如下结果：

```
namesAndDegrees.orderBy(desc("degree")).show()
...
+------------------+------+
|             topic|degree|
```

```
+------------------+------+
|          Research|  2596|
|           Disease|  1746|
|         Neoplasms|  1202|
|             Blood|   914|
|      Pharmacology|   882|
|      Tuberculosis|   815|
|        Toxicology|   694|
|      Drug Therapy|   678|
|     Jurisprudence|   661|
|Biomedical Research|   633|
```

这次分析中，大部分度数较高的顶点与前文讨论过的那些常见概念并没有什么不同，这一点在我们的意料之中。下一节我们将会使用 GraphX API 提供的新功能和一些经典的统计量，把那些没有什么意义的伴生二元组从图中过滤掉。

7.7　过滤噪声边

在当前的伴生图中，边的权重是基于一对概念同时出现在一篇论文中的频率来计算的。这种简单的权重机制的问题在于，它并没有对一对概念同时出现的原因加以区分，有时一对概念同时出现是因为它们具有某种值得我们关注的语义关系，但有时一对概念同时出现只是因为它们都频繁地出现在所有文档中，同时出现只是碰巧而已。我们需要使用一种新的权重机制，在给定概念在数据中的总体频繁度的情况下，它需要考虑给定的两个概念对于一个文档的“意义”或“新颖度”。我们将使用**皮尔逊卡方测试**（Pearson’s chi-squared test）来严格计算这种“意义”，也就是说，我们要测试一个概念的出现与其他概念的出现是否是独立的。

对任何概念对 A 和 B，我们可以建立一个 2 × 2 的列联表，它包含了这两个概念同时出现在 MEDLINE 文档中的次数。

	Yes B	No B	A Total
Yes A	YY	YN	YA
No A	NY	NN	NA
B Total	YB	NB	T

该表中 YY、YN、NY 和 NN 代表概念 A 和 B 在文档中出现 / 没出现的原始次数。YA 和 NA 是概念 A 的按行合计的出现次数，YB 和 NB 是概念 B 按列合计的出现次数，值 T 则是文档的总数。

卡方测试时，我们可以把 YY、YN、NY 和 NN 看成某个未知分布的观测，可以根据这些值计算**卡方统计量**：

$$\chi^2 = T \frac{(|YY * NN - YN*NY| - T/2)^2}{YA*NA*YB*NB}$$

请注意，这个卡方统计量的公式包括一个术语“- T/2”。这是耶茨的连续性校正（Yates's continuity correction，https://en.wikipedia.org/wiki/Yates's_correction_for_continuity），一些公式中并没有包含。

如果样本实际上是独立的，我们期望该统计量服从适当自由度的**卡方分布**。假定 r 和 c 是待比较的两个随机变量的基数，则自由度为 $(r - 1)(c - 1) = 1$。卡方统计量大则表明随机变量相互独立的可能性小，因此两个概念同时出现是有意义的。更具体地讲，自由度为 1 的卡方分布的 CDF（累积分布函数）给出一个 p 值，它是我们拒绝变量是独立的这个备择假设的置信水平。

本节将使用 GraphX 来计算伴生图中每个概念对的卡方统计量。

7.7.1 处理EdgeTriplet

求卡方统计量时最简单的部分就是计算 `T`，也就是需要考虑的文档的总个数。只要简单数一下 `medline` RDD 中的条目个数就可以轻松地得到这个 `T`，代码如下：

```
val T = medline.count()
```

计算每个概念在多少篇文档中出现也相对简单，本章前面建立 DataFrame 实例 `topicDist` 时已经讨论过，但我们现在需要将其表示为主题的散列值及其计数组成的 RDD：

```
val topicDistRdd = topicDist.map{
  case Row(topic: String, cnt: Long) => (hashId(topic), cnt)
}.rdd
```

有了这个表示主题词出现次数的 `VertexRDD`，就可以把它作为顶点集合，再加上已有的 `edges` RDD 一起用来创建一个新图：

```
val topicDistGraph = Graph(topicDistRdd, topicGraph.edges)
```

现在我们拥有计算 `topicCountGraph` 中每条边的卡方统计量所需的所有信息。计算卡方统计量，需要组合顶点数据（比如每个概念在一个文档中出现的次数）和边数据（比如两个概念同时出现在一个文档中的次数）。为了支持这种计算，GraphX 提供了一个数据结构 `EdgeTriplet[VD, ED]`，该数据结构将顶点和边的属性连同两个顶点的 ID 一起包装进一个对象。给定 `topicDistGraph` 上的一个 `EdgeTriplet`，就能算出卡方统计量，代码如下：

```
def chiSq(YY: Long, YB: Long, YA: Long, T: Long): Double = {
  val NB = T - YB
  val NA = T - YA
  val YN = YA - YY
  val NY = YB - YY
```

```
  val NN = T - NY - YN - YY
  val inner = math.abs(YY * NN - YN * NY) - T / 2.0
  T * math.pow(inner, 2) / (YA * NA * YB * NB)
}
```

然后可以用该方法通过 `mapTriplets` 算子转换边的值。`mapTriplets` 算子返回一个新图，这个图的边的属性就是每个伴生对的卡方统计量。于是我们就可以大概知道该统计量在所有边上的分布情况了：

```
val chiSquaredGraph = topicDistGraph.mapTriplets(triplet => {
  chiSq(triplet.attr, triplet.srcAttr, triplet.dstAttr, T)
})
chiSquaredGraph.edges.map(x => x.attr).stats()
...
(count: 213745, mean: 877.96,
 stdev: 5094.94, max: 198668.41,
 min: 0.0)
```

计算完卡方统计量，我们想用它去过滤那些没有意义的伴生概念对。从边的分布可以看出，数据中卡方统计量的范围很大，所以应该过滤掉更多的噪声边。对一个 2×2 的列联表，如果变量没有相关性，我们期望卡方指标的值服从自由度为 1 的卡方分布。自由度为 1 的卡方分布的第 99.999 百分位数大约为 19.5，因此我们将该值作为过滤边的阈值，这样过滤后图中就只剩下那些置信度**非常高**的有意义的伴生关系。我们将在图上利用 `subgraph` 方法进行过滤，这个方法接受 `EdgeTriplet` 的一个布尔函数，用以判断子图应该包含哪些边：

```
val interesting = chiSquaredGraph.subgraph(
  triplet => triplet.attr > 19.5)
interesting.edges.count
...
140575
```

我们采用非常严格的过滤规则，将原始伴生关系图中约三分之一的边都排除在外。该规则没有将更多的边过滤掉，这不是件坏事，因为我们预期图中大多数伴生概念实际上是语义相关的，并且它们因此同时出现的次数较多，而不是碰巧。下一节我们将分析子图的连接度和总体度分布，目的是了解去掉噪声边是否会对图的结构造成重大影响。

7.7.2　分析去掉噪声边的子图

我们先在子图上运行连通组件算法，并检查组件个数和组件大小，这里我们使用了本章前面为原始图编写的函数：

```
val interestingComponentGraph = interesting.connectedComponents()
val icDF = interestingComponentGraph.vertices.toDF("vid", "cid")
val icCountDF = icDF.groupBy("cid").count()
icCountDF.count()
```

```
...
878

icCountDF.orderBy(desc("count")).show()
...
+--------------------+-----+
|                 cid|count|
+--------------------+-----+
|-9218306090261648869|13610|
|-8193948242717911820|    5|
|-2062883918534425492|    4|
|-7016546051037489808|    3|
|-7685954109876710390|    3|
| -784187332742198415|    3|
| 1765411469112156596|    3|
| 2742772755763603550|    3|
|-8679136035911620397|    3|
```

过滤掉三分之一的边对图的连通性影响很小，也没有改变最大的连通组件的规模；删除图中的“无趣”边，使得主题图的总体连通性保持完好。检查一下过滤后图的度分布，情况也较为类似：

```
val interestingDegrees = interesting.degrees.cache()
interestingDegrees.map(_._2).stats()
...
(count: 13721, mean: 20.49,
 stdev: 29.86, max: 863.0, min: 1.0)
```

过滤前顶点度的平均值约为 31，过滤后顶点度的平均值稍微小一些，约为 20。然而更值得注意的是，过滤前后顶点的最大度下降非常大，过滤前为 2569，过滤后为 863。我们看一下过滤之后概念和度的关系，情况如下：

```
interestingDegrees.innerJoin(topicGraph.vertices) {
  (topicId, degree, name) => (name, degree)
}.values.toDF("topic", "degree").orderBy(desc("degree")).show()
...
+--------------------+------+
|               topic|degree|
+--------------------+------+
|            Research|   863|
|             Disease|   637|
|        Pharmacology|   509|
|           Neoplasms|   453|
|          Toxicology|   381|
|          Metabolism|   321|
|        Drug Therapy|   304|
|               Blood|   302|
|       Public Policy|   279|
|       Social Change|   277|
...
```

看起来我们的卡方过滤准则效果不错：它在清除对应普遍概念的边的同时，还保留了代表

概念之间有意义并且有值得注意的关系的那些边。我们可以继续用不同的卡方过滤准则进行试验，并且观察它们对图的连通性和度分布的影响。如果能找到卡方分布的某个值，并使用它作为过滤准则时，图中大型连通组件开始瓦解，这种尝试将是很有意义的。或者那个最大的组件只是不断“融化”，就像一座巨大的冰山随着时间慢慢消融。

7.8 小世界网络

图的连通性和度分布让我们了解了图的总体结构，而 GraphX 简化了这些属性的计算和分析。在这一节中我们将深入讲解 GraphX API 并介绍如何利用这些 API 来计算 GraphX 并不内置支持的图的一些高级属性。

随着计算机网络的崛起（比如网页，以及 Facebook 和 Twittert 等社交网络），数据科学家现在有了丰富的数据集。这些数据集描述了真实的网络结构和形态，而不是数学家和图论学家所研究的传统的理想模型。最早论述真实网络属性的论文之一就是 Duncan Watts 和 Steven Strogatz 于 1998 年发表的论文“Collective Dynamics of‘Small-World’Networks”（http://www.stevenstrogatz.com/articles/collective-dynamics-of-small-world-networks-pdf）。这篇会议论文第一次为具有两个“小世界”属性的图提出了数学生成模型。现实生活中的图具有如下两个“小世界”属性。

- 网络中大部分节点的度都不高，它们与其他节点形成相对稠密的簇。也就是说，一个节点的邻接点大部分也是相连的。
- 虽然图中大部分节点的度不高而且属于相对稠密的簇，但只需经过少数几条边可能从一个网络节点快速到达另一个节点。

对上述两个属性，Watts 和 Strogatz 分别定义了一个指标，这样就可以根据图的指标强度对图进行排序。本节我们将用 GraphX 来对我们的概念网络计算这些指标，并且将得到的指标和理想随机图的这些指标进行比较，从而测试我们的概念网络是否具有小世界的属性。

7.8.1 系和聚类系数

如果对每个顶点都存在一条边使其与其他任何节点都相连，则这个图就是**完备的**。给定一个图，可能有多个顶点子集是完备的，我们把这些完备的子图称为**系**（clique）。图中存在许多大型的系表示该图具有某种局部稠密结构，我们发现真实的小世界网络也具有这种局部稠密结构。

不幸的是，在给定图中寻找系是非常困难的。判断一个图是否有给定大小的系是一个 NP 完全问题。也就是说，即使在一个小型的图中寻找系，其计算复杂度也是非常高的。

计算机科学家提出了许多简单的指标，利用这些指标可以较好地了解一个图的局部稠密性，而不用花费巨大的计算代价来寻找给定大小的图中所有的系。其中一个指标就是顶点

的**三角计数**，**三角形**是一个完备图，顶点 V 的三角计数就是包含该顶点的三角形的个数。三角计数度量了 V 有多少个邻接点是相互连接的。Watts 和 Strogatz 定义了一个新的指标，称为**局部聚类系数**，它是一个顶点的实际三角计数与该顶点与其邻接点可能的三角计数的比率。对无向图来说，有 k 个邻接点和 t 个三角计数的顶点，其局部聚类系数 C 为：

$$C = \frac{2t}{k(k-1)}$$

现在我们用 GraphX 来计算过滤后的概念图的每个节点的局部聚类系数。GraphX 有一个内置方法 `triangleCount`，它返回一个 Graph 对象，其中 `VertexRDD` 包含了每个顶点的三角计数。

```
val triCountGraph = interesting.triangleCount()
triCountGraph.vertices.map(x => x._2).stats()
...
(count: 14548, mean: 74.66, stdev: 295.33, max: 11023.0, min: 0.0)
```

要计算局部聚类系数，我们需要通过每个顶点可能的三角计数，对该顶点的三角计数进行归一化。每个顶点可能的三角计数可以从 `interestingDegrees` RDD 计算得出，代码如下：

```
val maxTrisGraph = interestingDegrees.mapValues(d => d * (d - 1) / 2.0)
```

现在我们要把 `triCountGraph` 中包含三角计数的 `VertexRDD` 和上面得到的归一化 `VertexRDD` 进行联结，并计算二者的比率，在这个过程中，对那些只有一条边的顶点要注意避免零除问题：

```
val clusterCoef = triCountGraph.vertices.
  innerJoin(maxTrisGraph) { (vertexId, triCount, maxTris) => {
    if (maxTris == 0) 0 else triCount / maxTris
  }
}
```

对图中所有顶点局部聚类系数取平均值，就得到网络平均聚类系数：

```
clusterCoef.map(_._2).sum() / interesting.vertices.count()
...
0.30624625605188605
```

7.8.2 用Pregel计算平均路径长度

小世界网络的第二个属性就是任何两个节点之间的最短路径是短的，本节我们将计算过滤之后的概念图中的大型连通组件节点的平均路径长度。

计算图中顶点之间的路径长度是一个迭代过程，和我们之前寻找连通组件的迭代过程类似。该过程的每个阶段，每个顶点将保留它所接触过的顶点列表并记录到这些顶点的距

离。接着每个顶点都向其邻接点查询它对应的节点列表，如果发现该列表中有新的顶点，就用新节点更新自己的节点列表。查询邻接点并更新自己节点列表的过程一直继续下去，直到所有节点都没有发现有新节点需要添加为止。

这个在大规模分布式图上运行的以顶点为中心的迭代式并行计算方法，是以谷歌在 2009 年发表的题为"Pregel: a system for large-scale graph processing"的论文（https://dl.acm.org/citation.cfm?id=1807184）为基础的。Pregel 早于 MapReduce 之前就已经提出，它基于一个称为**批量同步并行**（Bulk-Synchronous Parallel，BSP）的分布式计算模型。BSP 程序将并行处理阶段分成两个步骤：**计算**和**通信**。在计算环节，图中每个顶点检查自己的内部状态并决定是否向图中其他节点发送**消息**。在通信环节，Pregel 框架负责将计算环节得到的消息路由到相应的顶点，目标顶点处理接收到的消息之后更新自己的内部状态，并可能在下一个计算环节中产生新消息。计算和通信的过程会一直继续下去，直到图中所有顶点都一致投票同意停止运行，这时整个过程就结束了。

BSP 是最早的并行编程模型之一，它具有良好的通用性，而且具有容错性，因此设计 BSP 系统时捕捉并保持任何计算阶段的系统状态是可能的。有了这些状态后，如果某台机器发生故障，就可以从其他机器上复制出发生故障的机器的状态，整个计算就可以回滚到故障发生前的状态，这样计算过程就可以继续下去。

自从谷歌发表了关于 Pregel 的论文之后，人们将 BSP 编程模型移植到 HDFS 之上并开发了许多开源项目，其中包括 Apache Giraph 和 Apache Hama。实践证明这些系统对那些适用于 BSP 编程模型的特定问题非常有用，比如大规模 PageRank 运算。但是由于这些项目难以集成到标准的数据并行处理工作流中，所以它们并没有广泛地被数据科学家当作分析工具，也就没有被广泛部署。而 GraphX 解决了这个问题。由于 GraphX 可以方便地使用图来描述数据并设计算法来对图进行处理，因此数据科学家可以轻松地将图计算集成到数据并行处理工作流中，而且 GraphX 还提供了表达 BSP 运算的内置 `pregel` 运算符，这个算子是以图为基础的。

本节我们将说明怎样使用这个运算符来实现对一个图的平均路径长度的计算，这是一个迭代式的图并行运算，包括：

(1) 分析出每个顶点需要记录的状态；
(2) 实现一个函数，需要考虑当前的状态，并且根据两个相连顶点决定下一阶段要发送哪些消息；
(3) 实现一个函数，汇总来自不同顶点的所有消息，然后将函数的结果传递给顶点以便更新其状态。

使用 Pregel 实现分布式算法时需要确定 3 个问题。第一，要确定用何种数据结构表示每个顶点状态和顶点之间传递的消息。对我们要解决的平均路径长度问题，我们希望每个顶点都有一个查询表，这个查询表包含当前顶点所知道的顶点的 ID 和它到这些顶点的距离。

我们将为每个顶点建立一个 Map[VertexId, Int] 并把这些信息存储在其中。类似地，发送给每个顶点的消息也应该有一个查询表，该表包含顶点 ID 和距离。这个距离是根据邻接点传递过来的信息计算出来的，同样可以用 Map[VertexId, Int] 来表示这些信息。

确定了顶点状态和消息内容的数据结构之后，我们可以实现两个函数。第一函数是 mergeMaps，用于将新消息中的信息合并到顶点状态之中。对我们讨论的问题来说，顶点状态和消息都是 Map[VertexId, Int] 类型的，因此需要把两个 map 中的内容合并在一起并将每个 VertexId 关联到两个 map 中该 VertexId 对应两个条目的最小值。

```
def mergeMaps(m1: Map[VertexId, Int], m2: Map[VertexId, Int])
  : Map[VertexId, Int] = {
  def minThatExists(k: VertexId): Int = {
    math.min(
      m1.getOrElse(k, Int.MaxValue),
      m2.getOrElse(k, Int.MaxValue))
  }

  (m1.keySet ++ m2.keySet).map {
    k => (k, minThatExists(k))
  }.toMap
}
```

顶点的 update 函数同样有一个 VertexId 参数，因此我们定义一个很简单的函数，它有一个 VertexId 类型参数和一个 Map[VertexId, Int] 类型参数，但所有实质性的任务由 mergeMaps 代理执行，代码如下：

```
def update(
    id: VertexId,
    state: Map[VertexId, Int],
    msg: Map[VertexId, Int]) = {
  mergeMaps(state, msg)
}
```

因为我们在算法执行过程中传递的消息的类型也是 Map[VertexId, Int]，并且我们想把这些消息合并起来并保留每个键对应的最小值，所以我们同样可以在 Pregel 运行的 reduce 阶段使用 mergeMaps 函数。

最后一步，通常也是最复杂的一步：我们需要编写代码来构建发送给每个顶点的消息，依据是每次迭代时每个顶点从邻接点收到的信息。这里的基本思想如下：每个顶点将当前的 Map[VertexId, Int] 中每个键对应的值加 1，然后用 mergeMaps 方法把加过 1 后的 map 值和从邻接点收到的值合并，如果 mergeMaps 方法的返回结果与邻接点内部的 Map[VertexId, Int] 不同，就将该结果发送给邻接点。这一系列操作的代码如下：

```
def checkIncrement(
    a: Map[VertexId, Int],
    b: Map[VertexId, Int],
    bid: VertexId) = {
```

```
    val aplus = a.map { case (v, d) => v -> (d + 1) }
    if (b != mergeMaps(aplus, b)) {
      Iterator((bid, aplus))
    } else {
      Iterator.empty
    }
  }
```

实现好 checkIncrement 之后，我们来定义 iterate 函数。在每个 Pregel 迭代过程中，我们用这个函数来对 EdgeTriplet 内部的 src 和 dst 顶点执行消息更新，代码如下：

```
def iterate(e: EdgeTriplet[Map[VertexId, Int], _]) = {
  checkIncrement(e.srcAttr, e.dstAttr, e.dstId) ++
  checkIncrement(e.dstAttr, e.srcAttr, e.srcId)
}
```

每次迭代时，需要根据每个顶点已经知道的路径长度来确定需要传递给每个顶点的路径长度。接着我们需要返回一个迭代器，它包含一个 (VertexId, Map[VertexId, Int]) 元组，其中第一个 VertexId 代表消息的目的顶点 ID，Map[VertexId, Int] 代表消息本身。

如果一次迭代中有任何顶点没有收到消息，pregel 运算符会认为该顶点的运算已经完成并将不再把它包括在后续处理中。一旦 iterate 方法没有消息发生给任何顶点，算法就结束。

> 相对于其他 BSP 系统实现（比如 Giraph），GraphX 的 pregel 运算符实现有一个限制：GraphX 中只有存在连接边的顶点之间才能发送消息，而 Giraph 可以在图中**任何**节点间发送消息。

现在完成了函数的编写，我们可以准备 BSP 运行所需的数据了。如果集群内存够多的话，我们可以用 GraphX 和 Pregel 式的算法计算任意两个顶点之间的路径长度。但是，对只想了解图中路径长度的总体分布而言，我们没必要这么做。其实我们只需要在顶点的一个随机样本子集上计算任意两个顶点之间的路径长度。使用 RDD 的 sample 方法可以对所有 VertexId 进行 2% 的不重复采样，随机数生成器的随机种子采用 1729L。

```
val fraction = 0.02
val replacement = false
val sample = interesting.vertices.map(v => v._1).
  sample(replacement, fraction, 1729L)
val ids = sample.collect().toSet
```

现在我们要建立一个新的 Graph 对象，如果一个顶点在样本子集中，Graph 对象的顶点 Map[VertexId, Int] 的值非空：

```
val mapGraph = interesting.mapVertices((id, _) => {
  if (ids.contains(id)) {
    Map(id -> 0)
  } else {
```

```
      Map[VertexId, Int]()
    }
  })
```

最后，为了触发算法的运行，我们需要向顶点发送一个初始消息。对我们讨论的算法而言，这个初始消息是一个空的 Map[VertexId, Int]。接下来我们就可以调用 pregel 方法，pregel 方法后面是在每次迭代中要执行的 update、iterate 和 mergeMaps 方法。

```
val start = Map[VertexId, Int]()
val res = mapGraph.pregel(start)(update, iterate, mergeMaps)
```

上面的代码可能需要运行几分钟。算法的迭代次数是样本集中最长路径的长度加 1。但代码运行结束后，我们可以用 flatMap 方法得到顶点的 (VertexId, VertexId, Int) 元组，它就是刚才算出来的不同路径的长度：

```
val paths = res.vertices.flatMap { case (id, m) =>
  m.map { case (k, v) =>
    if (id < k) {
      (id, k, v)
    } else {
      (k, id, v)
    }
  }
}.distinct()
paths.cache()
```

我们现在可以计算非零路径长度的概要统计量和样本路径长度的直方图：

```
paths.map(_._3).filter(_ > 0).stats()
...
(count: 3197372, mean: 3.63, stdev: 0.78, max: 8.0, min: 1.0)

val hist = paths.map(_._3).countByValue()
hist.toSeq.sorted.foreach(println)
...
(0,255)
(1,4336)
(2,159813)
(3,1238373)
(4,1433353)
(5,335461)
(6,24961)
(7,1052)
(8,23)
```

样本的平均路径长度为 3.63，上一节我们计算出了聚类系数为 0.306。表 7-1 给出了 3 个不同的小世界网络的平均路径长度和聚类系数，同时还给出了对这 3 个网络进行随机采样（顶点数和边数相同）之后的图的相应概要统计量。这些数据来源于 Auber 等于 2003 年发表的论文“Multiscale visualization of small world networks”（https://dl.acm.org/citation.

cfm?id=1947385)。

表7-1：小世界网络举例

图	平均路径长度（APL）	聚类系数（CC）	随机APL	随机CC
IMDB	3.20	0.967	2.67	0.024
macOS 9	3.28	0.388	3.32	0.018
.edu 网站	4.06	0.156	4.048	0.001

IMDB 图是根据参演同一部电影的演员生成的，macOS 9 网络描述的是 OS 9 中头文件被包含在相同源代码文件中的情况。第三行 .edu 网站是关于以顶级域名 .edu 结尾的网站相互链接的情况，引用源自 Adamic 1999 年的一篇论文（http://www.hpl.hp.com/research/idl/papers/smallworld/smallworldpaper.html）。我们的分析结果表明，MEDLINE 论文引用索引中的 MeSH 标签网络的平均路径长度和聚类系数值，与其他知名的小世界网络的对应统计量差不多。考虑到平均路径长度比较小，它们的聚类系数比我们预想的都要大。

7.9　小结

起初人们研究小世界网络只是出于好奇。现实世界的网络有如此多的种类，不管它们来自社会科学、政治学，还是来自神经科学和细胞生物学，都有非常相似而奇特的结构属性，这种现象非常有意思。最近的研究表明，当这些网络中的小世界结构出现异常时，就暗示着这个网络可能发生了功能性问题。杜克大学的 Jeffrey Petrella 博士收集了许多这方面的研究（http://pubs.rsna.org/doi/full/10.1148/radiol.11110380），这些研究表明大脑神经元网络具有小世界结构，这种小世界结构异常的病人被诊断出患有阿尔兹海默症、精神分裂症、抑郁症或注意力缺陷障碍。通常现实世界中的图都应该具有小世界属性，如果没有显示这种属性则表示可能存在问题，比如公司之间交易或信托关系的小世界图中可能反映出欺诈活动。

第 8 章

纽约出租车轨迹的空间和时间数据分析

作者：乔希·威尔斯

> 时间和空间最让我困惑；但也没什么么不让我那么困惑，因为我从来不想别的。
>
> ——Charles Lamb

纽约的黄色出租车很有名，对许多到纽约旅游的人来说，一边吃着街头小店买来的热狗，一边招呼黄色出租车，已经成为旅游中不可或缺的一部分，就像一定要坐电梯上帝国大厦顶层一样。

纽约本地人对探知何时何地最容易打到车可谓各怀绝技，特别是高峰期或下雨天。但在每天下午 4 点 ~ 5 点出租车换班的时段，估计像他们这样的高手也只能推荐你去坐地铁了。每天这个时候，黄色出租车需要回到调度中心（通常在皇后区）进行交班。如果交班晚了，司机可是要交罚款的。

2014 年 3 月，纽约市出租车及礼车委员会在其 Twitter 账号 @nyctaxi（https://twitter.com/nyctaxi）上公布了出租车的信息图。这个信息图可以显示任意时刻在途出租车的数量和在途出租车被占用的百分比。很显然，在下午 4 点 ~ 6 点，在途车数将大大减少，而且其中三分之二的车都被占用。

这条推文引起了城市规划专家 Chris Whong 的注意，Chris Whong 可是个数据迷，他立刻给 @nyctaxi 账号发了一条推文去了解信息图中所用的数据是否公开。委员会回复说只要提交一份 FOIL（Freedom of Information Law，信息自由法律）申请，并提供足够大的硬盘就

可以拿到他想要的数据。于是 Chris Whong 就填好 PDF 申请表，买了两块 500 GB 的硬盘并寄给委员会。两个工作日之后，Chris 就拿到了 2013 年 1 月 1 日至 12 月 31 日全年的所有出租车乘车数据。更令人高兴的是，Chris 甚至把所有运营数据也公布到了网上，这些数据成为很多漂亮的纽约交通信息图的依据。

出租车**利用率**，也就是出租车有乘客乘坐的时间与在途时间的比例，是理解出租车经济学的一个很重要的统计量。影响利用率的一个因素就是乘客的目的地：如果出租车乘客中午时分在联合广场附近下车，不出两分钟肯定又有乘客上车。但是乘客如果是凌晨两点在史坦顿岛下车，那这辆出租车就只能开回到曼哈顿才能接下一单生意。我们想对这种影响进行量化，并由此归结出租车平均等单时间与乘客下车点区域的函数关系，这些区域包括曼哈顿区、布鲁克林区、皇后区、布朗克斯区、史坦顿岛和其他区域（比如，乘客在纽约国际机场之类的市郊下车的情况）。

进行数据分析时我们往往要处理两类数据：**时间数据**（比如日期和时间）和**空间数据**（比如经纬度和边界）。自本书第 1 版问世以来，我们在 Spark 中使用时间数据的方式已经改进了很多，例如 Java 8 新发布的 `java.time` 包，以及 SparkSQL 从 Apache Hive 项目集成的 UDF，其中包括许多时间处理函数，例如 `date_add` 和 `from_timestamp`，这使得 Spark 2.*x* 在时间处理上比 Spark 1.*x* 更胜一筹。另一方面，空间数据仍然是一种相当专业的分析，需要依赖第三方库；并且为了能够在 Spark 中有效地处理这些数据，我们还要编写自己的 UDF。

8.1 数据的获取

为了分析出租车平均等单时间与乘客下车点区域的函数关系，我们只需要 2013 年 1 月以后的打车费用数据，这些数据解压之后大约有 2.5 GB，你可以从 http://www.andresmh.com/nyctaxitrips/ 下载 2013 年每个月的数据。如果你手头有一个足够大的 Spark 集群，也可以在全年的数据上重现接下来的分析。现在我们先在客户端机器上建立一个工作目录，然后看一下运营数据的结构，代码如下：

```
$ mkdir taxidata
$ cd taxidata
$ curl -O https://nyctaxitrips.blob.core.windows.net/data/trip_data_1.csv.zip
$ unzip trip_data_1.csv.zip
$ head -n 10 trip_data_1.csv
```

除了文件头，CSV 文件每行数据代表一次打车记录。每条打车记录包含如下信息：出租车（车牌号的散列）、司机（出租车司机驾照号的散列）、打车的开始时间和结束时间，以及乘客上车点和下车点的经纬度坐标。

8.2　基于Spark的第三方库分析

Java 平台的一大特点就是多年来用这个语言开发了大量代码：对于你可能用到的任何数据结构和算法，很可能早就有人写好了一个用于解决这个问题的 Java 库，而且很可能有个开源的版本可以下载来用，根本不用付什么许可费。

当然，我们不能因为有这样一个免费的库存在就一定要使用它，开源项目的质量参差不齐，bug 修复和新特性的进展状态也不一样，而且 API 设计和文档与教程的易用性也不相同。

我们选择工具的过程和开发人员为应用开发选择工具的过程不太一样。我们希望选择的工具便于交互式数据分析，并且易于分布式应用使用。具体来说，我们需要确保在 RDD 中要处理的主要数据类型支持 `Serializable` 接口，最好还能方便地用 Kryo 之类的库进行序列化。

除此之外，我们还希望用于交互式分析的库的外部依赖越少越好。虽然 Maven 和 SBT 之类的工具可以帮助应用开发人员在构建引用时处理复杂的依赖关系，但我们希望用一个 JAR 文件搞定所有代码，只要把这个 JAR 加载到 Spark shell 中就可以开始数据分析了。然而，在 Spark 中载入许多依赖库可能会导致它们与 Spark 自身所依赖的其他库之间的版本冲突，这种错误非常难调试，被开发人员称为 JAR **灾难**。

最后，我们希望工具的 API 相对简单而且功能丰富，不需要使用那些花哨的面向 Java 的设计模式，比如什么抽象工厂和访问者之类的模式。虽然这些模式对应用开发人员可能很有用，但它们往往在代码中引入与分析无关的复杂度。Scala 提供了对许多 Java 工具的封装，利用这些封装可以在使用 Scala 时减少设计模式所需的程式化代码，如果我们的工具也能这样那就更好了！

8.3　基于Esri Geometry API和Spray的地理空间数据处理

在 Java 8 中使用时间数据变得非常容易，这要感谢 `java.time` 包，它是基于非常成功的 JodaTime 库设计的。对地理空间数据来说，问题就没这么简单了。这是因为有非常多不同的工具和库，这些工具和库又有不同的功能、开发状态和成熟度，目前并没有一个主流的 Java 库对所有地理空间应用场景都适用。

选择类库时，首先要确定需要处理哪种空间数据。地理数据主要分为矢量和光栅两种，每一种都有不同的处理工具。在本章的场景中，我们有出租车乘客上车点和下车点的经纬度数据，以及表示纽约各个区边界的矢量数据，这些矢量数据用 GeoJSON 格式存储。因此

我们需要一个可以解析 GeoJSON 数据并能处理其空间关系的工具。具体来说，就是该工具可以判断某经纬度所代表的点是否在某个区边界所组成的多边形中。

不幸的是，目前没有一个开源的库正好能满足我们的要求。有一个 GeoJSON 的解析工具可以把 GeoJSON 转换成 Java 对象，但没有相关的地理空间工具能对转换得到的对象进行空间关系分析。有一个名叫 GeoTools 的项目，但它的组件和依赖关系实在太多，我们不希望在 Spark shell 中选用有太多复杂依赖的工具。最后有一个 Java 版本的 Esri Geometry API，它的依赖很少而且可以分析空间关系，但它只能解析 GeoJSON 标准的一个子集，因此我们必须对下载的 GeoJSON 数据做一些预处理。

对数据分析师而言，没有工具就无法解决问题。但我们可是数据科学家啊！如果手头的工具解决不了问题，那我们就自己创建一个新工具。对本章要解决的问题而言，为了解析所有的 GeoJSON 数据，我们要加入新的 Scala 功能，其中包括 Scala Esri Geometry API 不能处理的功能。有许多 Scala 项目提供 JSON 数据解析功能，我们使用其中一个来实现这些功能。接下来几节中的代码可以在本书的 GitHub 资料库（https://github.com/jwills/geojson）上找到，这些 Scala 代码可以用于任何地理空间分析项目。

8.3.1 认识Esri Geometry API

Esri 库的核心数据类型是 `Geometry` 对象，一个 `Geometry` 代表一个形状和它所在的地理位置，Esri 提供了一组空间分析操作用于分析几何图像及其关系。

这些操作包括计算几何图形的面积、判断两个图形是否重叠和求两个图形相加所得到的几何图形。

对本章示例来讲，我们有表示出租车乘客下车地点（经度和纬度）的几何图形，也有表示纽约市行政区域范围的几何图形。我们想知道它们的包含关系：一个给定的位置点是否在曼哈顿区对应的多边形里边？

Esri API 有一个助手类 `GeometryEngine`，它提供了执行所有空间关系操作的静态方法，其中就包括 `contains` 操作。`contains` 方法有 3 个参数：两个 `Geometry` 实例参数和一个 `SpatialReference` 实例参数。`SpatialReference` 实例参数表示用于地理空间计算的坐标系统。为了提高精度，我们需要分析地球球体上的点映射到二维坐标系统后相对于坐标平面的空间关系。地理空间工程师有一套标准的通用标识符（well-known identifier，称为 WKID），是一套最常用的坐标系统。这里我们将采用 WKID 4326，它也是 GPS 所用的坐标系统。

作为 Scala 开发人员，我们总是想方设法减少在 Spark shell 中进行交互式数据分析时输入的代码量。在 Spark shell 中可不像 Eclipse 和 IntelliJ 那样能自动为我们补全长方法名，也不能像这些开发环境一样提供语法糖来辅助看懂某种操作。根据 NScalaTime 库（它定义

了包装类 RichDateTime 和 RichDuration）的命名规范，我们将定义自己的 RichGeometry 类，它扩展了 Esri Geometry 对象并提供一些有用的辅助方法，代码如下：

```
import com.esri.core.geometry.Geometry
import com.esri.core.geometry.GeometryEngine
import com.esri.core.geometry.SpatialReference

class RichGeometry(val geometry: Geometry,
    val spatialReference: SpatialReference =
      SpatialReference.create(4326)) {
  def area2D() = geometry.calculateArea2D()

  def contains(other: Geometry): Boolean = {
    GeometryEngine.contains(geometry, other, spatialReference)
  }

  def distance(other: Geometry): Double = {
    GeometryEngine.distance(geometry, other, spatialReference)
  }
}
```

我们将为 Geometry 定义一个伴生对象，它可以将 Geometry 类实例隐式转换为 RichGeometry 类型：

```
object RichGeometry{
  implicit def wrapRichGeo(g: Geometry) = {
    new RichGeometry(g)
  }
}
```

记住，要想这种转换起作用，需要在 Scala 环境中导入这个隐式函数定义，代码如下：

```
import RichGeometry._
```

8.3.2 GeoJSON简介

表示纽约市行政区域范围的数据是 GeoJSON 格式的，GeoJSON 中核心的对象称为**特征**，特征由一个 geometry 实例和一组称为**属性**（property）的键 – 值对组成。其中 geometry 可以是点、线或多边形。一组特征称为 FeatureCollection。现在我们把纽约市行政区地图的 GeoJSON 数据下载下来，然后看看它的结构。

将数据下载到客户端机器上的 taxidata 目录，并将文件名改短：

```
$ curl -O https://nycdatastables.s3.amazonaws.com/2013-08-19T18:15:35.172Z/
  nyc-borough-boundaries-polygon.geojson
$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson
```

打开文件然后观察一下特征记录，注意属性和几何图形对象。对本章示例而言，也就是表示行政区域范围的多边形和保护行政区域名称及其他相关信息的属性。

可以用 Esri Geometry API 解析每个特征内部的几何 JSON，但这个 API 不能帮我们解析 id 或 properties 字段，properties 字段可能是任何 JSON 对象。为了解析这些对象，要用到 Scala 的 JSON 库，这样的库有很多。

这里正好可以用 Spray，它是一个用 Scala 构建 Web 服务的开源工具包。通过隐式调用 spray-json 的 toJson 方法，可以将任何 Scala 对象转换成相应的 JsValue。也可以通过调用它的 parseJson 方法将任何 JSON 格式的字符串转换成一个中间类型，然后在中间类型上调用 convertTo[T] 将其转换成一个 Scala 类型 T。Spray 内置了对常用 Scala 原子类型、元组和集合类型的转换实现，同时也提供了一个格式化工具，该工具可以定义自定义类型（比如 RichGeometry）与 JSON 之间相互转换的规则。

首先为表示 GeoJSON 的特征将建立一个 case 类。根据规范，特征是一个 JSON 对象，它必须有一个 geometry 字段和一个 properties 字段。geometry 代表 GeoJSON 的几何类型，properties 则是一个 JSON 对象，可以包含任意个数和类型的键 – 值对。特征也可以有一个可选字段 id，表示任何 JSON 标识符。我们的 case 类的 Feature 将为每个 JSON 字段定义相应的 Scala 字段，同时它还提供了在属性 map 中查找值的辅助方法：

```
import spray.json.JsValue

case class Feature(
    val id: Option[JsValue],
    val properties: Map[String, JsValue],
    val geometry: RichGeometry) {
  def apply(property: String) = properties(property)
  def get(property: String) = properties.get(property)
}
```

我们使用 RichGeometry 类实例来表示 Feature 中的 geometry 字段。我们通过 Esri Geometry API 的 GeoJSON 图形解析函数创建 RichGeometry 类实例。

还需要为 GeoJson FeatureCollection 定义一个 case 类。为了使 FeatureCollection 类更易于使用，实现 apply 和 length 这两个抽象方法，就能让它扩展 IndexedSeq[Feature] 这个 trait。这样就能直接在 FeatureCollection 实例上调用标准的 Scala Collections API 的方法，比如 map、filter 和 sortBy 等，而不用访问底层的 Array[Feature]。

```
case class FeatureCollection(features: Array[Feature])
    extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length
}
```

在定义表示 GeoJSON 数据的 case 类之后，还需要定义领域对象（RichGeometry、Feature 和 FeatureCollection）与相应的 JsValue 实例之间相互转换的格式。为此要创建 Scala 的单例对象，这些对象扩展了 RootJsonFormat[T] trait，这个 trait 定义了抽象方法 read(jsv:

JsValue): T 和 write(t: T): JsValue。对于 RichGeometry 类，我们可以将大部分的解析和格式化逻辑委派给 Esri Geometry API，也就是 GeometryEngine 类的 geometryToGeoJson 和 geometryFromGeoJson 方法。但对我们定义的 case 类，我们需要自己编写格式化逻辑。下面是 Feature 这个 case 类的格式化代码，其中包含了一些为处理可选字段 id 的特殊逻辑：

```
implicit object FeatureJsonFormat extends
    RootJsonFormat[Feature] {
  def write(f: Feature) = {
    val buf = scala.collection.mutable.ArrayBuffer(
      "type" -> JsString("Feature"),
      "properties" -> JsObject(f.properties),
      "geometry" -> f.geometry.toJson)
    f.id.foreach(v => { buf += "id" -> v})
    JsObject(buf.toMap)
  }

  def read(value: JsValue) = {
    val jso = value.asJsObject
    val id = jso.fields.get("id")
    val properties = jso.fields("properties").asJsObject.fields
    val geometry = jso.fields("geometry").convertTo[RichGeometry]
    Feature(id, properties, geometry)
  }
}
```

FeatureJsonFormat 对象中的 implicit 关键字是为了 Spray 库可以在 JsValue 实例上调用 convertTo[Feature] 时进行查找。可以在 GitHub 上找到 GeoJSON 库实现 RootJsonFormat 的其余源代码。

8.4　纽约市出租车客运数据的预处理

现在我们手头上有了 GeoJSON 和 JodaTime 库，该开始用 Spark 对纽约市出租车客运数据进行交互式分析了！先在 HDFS 上建立一个 taxidata 目录，并将载客数据复制到集群上：

```
$ hadoop fs -mkdir taxidata
$ hadoop fs -put trip_data_1.csv taxidata/
```

现在启动 Spark shell，使用 --jars 参数，将要用到的库导入 REPL 中：

```
$ mvn package
$ spark-shell --jars ch08-geotime-2.0.0-jar-with-dependencies.jar
```

Spark shell 加载完成后，就可以像在其他章中一样，用出租车数据创建一个数据集，并且检查一下前几行的数据：

```
val taxiRaw = spark.read.option("header", "true").csv("taxidata")
taxiRaw.show()
```

出租车数据看起来是一个格式良好的 CSV 文件，具有明确定义的数据类型。在第 2 章中，我们使用 spark-csv 中内置的类型推断库，自动将 CSV 数据从字符串转换为列的特定类型，这种自动转换的成本是双重的。第一，需要额外遍历一次数据，以便转换器可以推断出每一列的类型。第二，即使只想分析数据集中列的一个子集，也需要花费额外的资源，来推断那些不会用到的列的类型。对于较小的数据集而言，比如我们在第 2 章中分析的记录关联数据，额外的推断开销可以忽略不计。但是对于计划多次反复分析的超大数据集，花些时间编写代码，对确定需要的列进行自定义类型转换，这样做是值得的。本章将通过自定义代码自行完成转换。

我们来定义一个 case 类，它包含了分析时我们要用到的每条打车记录信息。由于我们要使用这个样例类作为 Dataset 的基础，需要注意的一点是，Dataset 只能针对一小部分数据类型进行优化，包括字符串、原子类型（像 Int、Double 等）和某些特殊的 Scala 类型（如 Option）。如果要利用 Dataset 类提供的性能增强和实用分析工具，我们的样例类只能包含属于这些类型的字段。[请注意，下面的代码清单仅仅是完整代码的说明性摘录，如果需要执行本章的所有代码，请参阅附带的第 8 章的源代码库（https://github.com/sryza/aas/tree/master/ch08-geotime），特别是 GeoJson.scala。]

```
case class Trip(
  license: String,
  pickupTime: Long,
  dropoffTime: Long,
  pickupX: Double,
  pickupY: Double,
  dropoffX: Double,
  dropoffY: Double)
```

我们将 pickupTime 和 dropoffTime 字段表示为长整型，其值代表从 Unix 纪元以来的毫秒数，并将各个上下车位置的 xy 坐标存储在单独的字段中，即使我们通常处理坐标值的方式是将坐标转换成 Esri API 中 Point 类的实例。

为了将 taxiRaw 数据集中的 Row 对象解析为 case 类的实例，需要创建一些辅助对象和函数。首先，需要注意数据中某些行的某些字段很可能是缺失的，因此当我们从 Row 对象中访问它们之前，需要先确认它们是否为空，否则程序会抛出一个错误。我们可以编写一个小的帮助类来解决这个问题。RichRow 类可以处理我们需要解析的任何一种 Row 对象：

```
class RichRow(row: org.apache.spark.sql.Row) {
  def getAs[T](field: String): Option[T] = {
    if (row.isNullAt(row.fieldIndex(field))) {
      None
    } else {
      Some(row.getAs[T](field))
    }
  }
}
```

在 RichRow 类中，getAs[T] 方法总是返回一个 Option[T]，而不是直接返回原始值。如果返回原始值，我们可以明确处理解析一个 Row 对象时出现的缺少字段的情况。在这个例子中，数据集中的所有字段都是字符串，所以我们只需要处理 Option[String] 类型的值。

接下来，我们需要使用 Java 的 SimpleDateFormat 类的实例来处理上下车时间，并使用适当的格式化字符串来获取以毫秒为单位的时间：

```
def parseTaxiTime(rr: RichRow, timeField: String): Long = {
  val formatter = new SimpleDateFormat(
       "yyyy-MM-dd HH:mm:ss", Locale.ENGLISH)
  val optDt = rr.getAs[String](timeField)
  optDt.map(dt => formatter.parse(dt).getTime).getOrElse(0L)
}
```

然后，我们将使用 Scala 的隐式 toDouble 方法解析上下车位置的经度和纬度，将 String 隐式地转为 Double，如果坐标缺失，则使用默认值 0.0：

```
def parseTaxiLoc(rr: RichRow, locField: String): Double = {
  rr.getAs[String](locField).map(_.toDouble).getOrElse(0.0)
}
```

把这些函数放在一起，得到如下的 parse 方法：

```
def parse(row: org.apache.spark.sql.Row): Trip = {
  val rr = new RichRow(row)
  Trip(
    license = rr.getAs[String]("hack_license").orNull,
    pickupTime = parseTaxiTime(rr, "pickup_datetime"),
    dropoffTime = parseTaxiTime(rr, "dropoff_datetime"),
    pickupX = parseTaxiLoc(rr, "pickup_longitude"),
    pickupY = parseTaxiLoc(rr, "pickup_latitude"),
    dropoffX = parseTaxiLoc(rr, "dropoff_longitude"),
    dropoffY = parseTaxiLoc(rr, "dropoff_latitude")
  )
}
```

可以用 taxiRaw 的前几条数据来测试 parse 函数，以此来验证它是否能正确处理样本数据。

8.4.1 大规模数据中的非法记录处理

实际处理过大规模数据集的人都知道，这些数据集中总有几条记录的格式不满足代码的要求。许多 MapReduce 作业和 Spark 处理管道会因为无法正常解析非法记录而抛出异常，致使运行失败。

我们可以逐个排除这些异常，先检查任务日志，然后分析抛出异常的每行代码，再修改代码以忽略或修正非法记录。这个过程相当漫长乏味，而且常常像是在玩打鼹鼠游戏：刚解决好一个异常，分区中后面的某条记录上又冒出了另一个异常。

有经验的数据科学家在处理新数据集时常用的一个策略就是在解析代码中增加一个 try-catch 块，这样任何非法记录就都可以写入到日志中而不会导致整个作业失败。如果整个数据集中只有几条非法记录，忽略掉这些记录并继续分析应该没问题。有了 Spark，我们甚至可以做得更好：通过调整解析代码，可以对数据中的非法记录进行交互式分析，就和其他分析一样轻松。

对 RDD 或数据集中的每条记录，解析代码的结果可能有两个：要么成功解析并返回一条有意义的结果，要么失败并抛出异常。抛出异常时我们希望得到非法记录本身和所抛出的异常。当操作结果有两种互斥的结果时，可以使用 Scala 的 Either[L, R] 类型来表示操作的返回类型。对我们本章的问题来说，L（left）结果代表成功解析得到的记录，而 R（right）结果是一个由异常和引起异常的记录组成的二元组。

safe 函数接受一个输入类型为 S => T 的参数 f 并且返回一个新的 S => Either[T, (S, Exception)] 类型，它会返回调用 f 的结果，或者在抛出异常时返回包含非法输入值和异常本身组成的元组：

```
def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {
  new Function[S, Either[T, (S, Exception)]] with Serializable {
    def apply(s: S): Either[T, (S, Exception)] = {
      try {
        Left(f(s))
      } catch {
        case e: Exception => Right((s, e))
      }
    }
  }
}
```

现在可以通过向 safe 函数传入 parse 函数（类型为 String => Trip）来得到一个更加安全的包装函数 safeParse，然后在 taxiRaw 数据集底层的 RDD 上调用 safeParse：

```
val safeParse = safe(parse)
val taxiParsed = taxiRaw.rdd.map(safeParse)
```

（注意，不能直接将 safeParse 应用于 taxiRaw 数据集，因为 Dataset API 不支持 Either[L, R] 类型。）

如果想要知道输入行中成功解析的记录数量，可以用 Either[L, R] 的 isLeft 方法，并结合使用 countByValue 这个动作：

```
taxiParsed.map(_.isLeft).
countByValue().
foreach(println)
...
(true,14776615)
```

运气真好——在记录解析过程中没有抛出异常！我们现在可以通过获取 Either 值左边的元素，将 taxiParsed RDD 转换为 Dataset[Trip] 实例：

```
val taxiGood = taxiParsed.map(_.left.get).toDS
taxiGood.cache()
```

即使 taxiGood 数据集中的记录解析正确，它们也还可能存在数据质量问题，这些问题有待我们进一步发现和处理。为了找出这些遗留的数据质量问题，我们要思考每条正确的乘车记录都应该满足的期望条件。

考虑到乘车数据的时间特性，任何乘车记录的下车时间都比上车时间晚是一个合理的规则。同样我们可以期望乘车时间不超过几个小时，虽然确实有可能存在这样耗时较长的乘车记录，比如高峰期打车或遇到事故延误的情况时打车要几个小时是可能的。将“合理”的打车时间的阈值设为多少合适，我们对此不是很确定。

我们来定义一个辅助方法 hours，它使用 Java 的辅助方法 TimeUnit，将上下车时间的差值从毫秒转换为小时：

```
val hours = (pickup: Long, dropoff: Long) => {
  TimeUnit.HOURS.convert(dropoff - pickup, TimeUnit.MILLISECONDS)
}
```

我们希望能够使用我们的 hours 函数，计算持续给定小时数以上行程的直方图分布。Dataset API 和 Spark SQL 就是为这种计算而设计的，但是默认情况下，我们只能在 Dataset 实例的列上使用这些方法，而 hour UDF 是从 pickupTime 和 dropoffTime 这两列中计算出来的。我们需要一种机制，将 hours 函数应用于数据集中的列，然后执行我们熟悉的正常过滤和分组操作。

这正是设计 Spark SQL UDF 的目标用例。通过在 Spark 的 UserDefinedFunction 类的实例中包装 Scala 函数，我们可以将该函数应用于 Dataset 中的列并分析结果。首先将 hours 封装到 UDF 中，然后计算我们的直方图：

```
import org.apache.spark.sql.functions.udf
val hoursUDF = udf(hours)
taxiGood.
  groupBy(hoursUDF($"pickupTime", $"dropoffTime").as("h")).
  count().
  sort("h").
  show()
...
+---+--------+
|  h|   count|
+---+--------+
| -8|       1|
|  0|22355710|
|  1|   22934|
```

```
|  2|     843|
|  3|     197|
|  4|      86|
|  5|      55|
...
```

现在看起来都不错，只有一条打车记录除外，它的乘车时间为 −8 小时！难道是电影《回到未来》中德洛雷安在纽约开出租挣外快？让我们来一探究竟：

```
taxiGood.
  where(hoursUDF($"pickupTime", $"dropoffTime") < 0).
  collect().
  foreach(println)
```

这给出了那条奇怪的记录，打车开始于 1 月 25 日下午 6 点，在同一天上午 10 点前结束。我们看不出来这条打车记录到底哪里出了错。但是由于看起来只有一条记录是这种情况，直接将它从记录中去掉应该没问题。

现在观察剩余的那些小时数大于零的记录，绝大多数出租车乘坐记录看起来不超过 3 个小时。我们将对 `taxiGood` RDD 进行过滤，只需要关心“典型”的乘坐记录的分布而暂时忽略那些异常情况。因为在 Spark SQL 中表示这个过滤条件要容易一些，所以用“hours”这个名字到 Spark SQL 中注册我们的 `hours` 函数，以便在 SQL 表达式中使用它：

```
spark.udf.register("hours", hours)
val taxiClean = taxiGood.where(
  "hours(pickupTime, dropoffTime) BETWEEN 0 AND 3"
)
```

UDF，用还是不用？

通过 Spark SQL，很容易将业务逻辑嵌入到标准 SQL 可用的函数中，就像我们对 `hours` 函数所做的那样。基于这一考虑，你可能会认为将所有业务逻辑转换为 UDF 是一个好主意，这样做也便于重用、测试和维护。但是，在代码中大量使用 UDF 之前，需要了解一些关于 UDF 的注意事项。

首先，UDF 对 Spark 的 SQL 查询计划器和执行引擎是不透明的，而标准的 SQL 查询语法却是透明的。因此，使用 UDF 而不是直接写 SQL 表达式，可能会影响查询性能。

其次，在 Spark SQL 中处理空值很快变得复杂起来，特别是对于处理多个参数的 UDF。编写的 UDF 若要正确地处理空值，需要使用 Scala 的 `Option[T]` 类型，或者使用 Java 包装类型（如 `java.lang.Integer` 和 `java.lang.Double`），而不是 Scala 中的基本类型 `Int` 和 `Double`。

8.4.2　地理空间分析

现在我们从地理空间角度来检查出租车数据。对每次乘车记录，我们各用一对经纬度来表

示乘客的上车地点和下车地点。我们想确定这两对经纬度分别属于哪个行政区，并且要找出那些起点不在纽约 5 个行政区之内的记录。比如，如果是从曼哈顿打车到纽约国际机场，这条记录应该是合法的，虽然它的终点并不在 5 个行政区之内。但如果打车的终点是南极，我们就有理由相信记录是非法的并且应该将其排除在分析之外。

为了分析行政区，需要将前面下载的 GeoJSON 数据加载并存储在 nyc-boroughs.geojson 文件中。scala.io 包中的 Source 类可以帮助我们轻松把文本文件或 URL 中的数据作为一个 String 读取到客户端中：

```
val geojson = scala.io.Source.
  fromFile("nyc-boroughs.geojson").
  mkString
```

现在需要用到本章前面讨论过的 GeoJSON 解析工具，通过在 Spark shell 中使用 Spray 和 Esri，就可以将 geojson 解析为 FeatureCollection case 类：

```
import com.cloudera.datascience.geotime._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]
```

我们建立一个简单的地点来测试 Esri Geometry API 的功能，并验证该 API 能正确找出指定坐标的所属行政区：

```
import com.esri.core.geometry.Point
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
```

在使用出租车乘车数据上的 features 之前，要思考一下怎样组织地理空间数据最有效。一个方法是研究专门为地理空间查询而优化的数据结构，比如四叉树，然后编写自己的实现代码。但我们先看一下是否能想出一个快速的启发式算法以省掉这部分工作。

find 方法将遍历 FeatureCollection 直到找到一个图形包含给定经纬度 Point 的特征为止。大部分打车记录的上车点和下车点都在曼哈顿地区，因此如果代表曼哈顿的地理空间特征在集合中早点出现，大部分的 find 方法调用将可以较快地返回。我们可以把每个特征的 boroughCode 属性作为排序的键，1 代表曼哈顿，5 代表史坦顿岛。对每个行政区特征内部，我们希望最大的多边形相关的特征排在较小的多边形之前，因为打车时大部分起点或终点会落在每个行政区中“较大”的地区。然后根据新政区代码和每个特征的几何图形的 area2D() 大小来对特征进行排序应该是个不错的策略：

```
val areaSortedFeatures = features.sortBy(f => {
  val borough = f("boroughCode").convertTo[Int]
  (borough, -f.geometry.area2D())
})
```

注意这里是根据 area2D() 取负值之后排序的，因为我们想让最大的多边形排在最前面，而 Scala 默认是从小到大排序。

现在可以将 areaSortedFeatures 序列中排好序的特征广播到集群上，然后写一个函数，利用这些特征来判断下车点落在 5 个行政区的哪一个中。

```
val bFeatures = sc.broadcast(areaSortedFeatures)

val bLookup = (x: Double, y: Double) => {
  val feature: Option[Feature] = bFeatures.value.find(f => {
    f.geometry.contains(new Point(x, y))
  })
  feature.map(f => {
    f("borough").convertTo[String]
  }).getOrElse("NA")
}
val boroughUDF = udf(bLookup)
```

我们可以在 taxiClean RDD 中的打车记录上应用 boroughUDF，创建一个按行政区统计的直方图。

```
taxiClean.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
...
+-----------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+-----------------------+--------+
|                 Queens|  672192|
|                     NA| 7942421|
|               Brooklyn|  715252|
|          Staten Island|    3338|
|              Manhattan|12979047|
|                  Bronx|   67434|
+-----------------------+--------+
```

像我们预期的那样，绝大多数打车记录的终点在曼哈顿地区，在史坦顿岛的相对较少。终点落在 5 个行政区之外的乘车记录数有点让人吃惊，而 NA 记录的数量比终点在布朗克斯区的记录数要多得多。现在我们从数据中取出几条这种记录：

```
taxiClean.
  where(boroughUDF($"dropoffX", $"dropoffY") === "NA").
  show()
```

打印出这些记录，会发现它们大部分的起点和终点都落在点 (0.0， 0.0) 上，表明这些记录的起点和终点数据缺失。由于这些数据对我们的分析帮助不大，应该把这种例外情况过滤掉：

```
val taxiDone = taxiClean.where(
  "dropoffX != 0 and dropoffY != 0 and pickupX != 0 and pickupY != 0"
).cache()
```

现在重新在 `taxiDone` RDD 上运行分析，得到如下结果：

```
taxiDone.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
...
+-----------------------+--------+
|UDF(dropoffX, dropoffY)|   count|
+-----------------------+--------+
|                 Queens|  670912|
|                     NA|   62778|
|               Brooklyn|  714659|
|          Staten Island|    3333|
|              Manhattan|12971314|
|                  Bronx|   67333|
+-----------------------+--------+
```

过滤掉起点或终点为零的记录后，5 个行政区的输出记录只是减少了一些，但 NA 对应的记录大部分被去掉了，剩下的那些终点落在郊区的记录条数现在看起来比较合理了。

8.5 基于Spark的会话分析

前面提到的一个目标是要研究出租车乘客下车区域与出租车等待下一单生意的等待时间之间的关系。现在 `taxiDone` 数据集包含了每个出租车司机的所有载客数据，但这些记录分布在不同的分区中。要计算一次载客结束到下次载客开始的时间间隔，需要把一个班次中的所有载客记录按一个司机一条记录进行汇总，然后把该班次中的载客记录按时间排序。排序让我们可以比较一次载客记录的下车时间和下一次载客的上车时间。这种对单个实体在不同时间的一系列事件的分析称为**会话分析**（sessionization），它经常用于对 Web 日志做网站用户行为分析。

会话分析是发掘数据价值和开发数据产品的一种非常强大的技术，可以帮助人们更好地进行决策。比如谷歌的自动拼写纠正引擎就是基于用户活动会话构建的。谷歌将每天在其网站上发生的每个事件（搜索、点击、地图访问等）用日志记录下来，并在这些记录上构建会话。为了找出可能的拼写纠正项，谷歌对这些会话进行处理并找出如下描述的情形：用户输入查询却没有做任何点击，几秒钟以后该用户又输入一个稍微不同的查询，然后点击查询结果，然后就离开谷歌了。找到上述情形之后，谷歌计算每两个这样的查询的模式出现的次数，如果次数足够频繁（比如如果我们发现每次输入查询“untied stats”几秒后输入查询“united states”），那么就可以假设第二个查询是第一个查询的拼写纠正项。

这个分析利用日志中展现出的人类行为模式来构建拼写纠正引擎，这个引擎比任何基于字

典的引擎都要强大得多。该引擎可以用于任何语言的拼写检查，并且可以用于纠正那些没有在任何字典中出现的词（比如某个创业公司的名字），甚至可以用于纠正类似“untied stats”这样两个单词拼写错误的查询。谷歌给出推荐搜索项和相关搜索项时也使用了类似的技术，并且将这个技术用于确定哪些查询应该返回一个 OneBox 结果。OneBox 类型的搜索结果直接显示在查询页面上，这样用户就不需要继续点击进入不同页面。OneBox 已经应用到谷歌天气、体育赛事得分、地址和许多其他类型的查询中。

现在每个实体发生的所有事件是散布在 RDD 的各个分区中的，因此我们需要按时间顺序将相关时间放在一起。下一节将演示如何使用 Spark 2.0 中引入的高级分析功能来高效地构造和分析会话。

构建会话：基于Spark的二级排序

在 Spark 中创建会话，最简单的方法就是根据标识符做 `groupBy`，然后根据时间戳标识符对打乱次序后的事件数据排序。如果每个实体只有少数事件，这种方法还是比较行得通的。然而，这个方法的扩展能力十分有限，因为它需要将每个实体的所有事件同时都放入内存，因此随着每个实体的事件数量越来越大，所占用的内存将会越来越大。我们需要一种构建会话的方法，它不需要在排序时将一个实体的所有事件同时放入内存。

在 MapReduce 中，可以通过**二级排序**（secondary sort）来构建会话，做法是创建一个由标识符和时间戳组成的组合键，根据该组合键对所有记录排序，然后用一个定制的分区器（partitioner）和分组函数保证相同标识符对应的所有记录都在同一个结果分区中。幸运的是，Spark 也支持这种排序模式，为此我们可以使用 Spark 的 `repartition` 和 `sortWithinPartitions` 转换。在 Spark 2.0 中，对一个数据集进行会话处理 , 只需 3 行代码：

```
val sessions = taxiDone.
  repartition($"license").
  sortWithinPartitions($"license", $"pickupTime")
```

首先，使用 `repartition` 方法，确保 `Trip` 记录中所有 `license` 列值相等的记录被分配在同一个分区中。然后，在每个分区中，按照 `license` 的值对记录进行排序（因此同一个司机的所有行程都在一起），再通过 `pickupTime` 进行排序，完成后每个分区中的行程都是排好序的。现在，当我们使用像 `mapPartitions` 这样的方法处理行程记录时，可以肯定，这些行程是以会话分析的最佳方式排序的。由于这个操作会触发 shuffle 和一点计算，而且需要多次使用这个结果，我们应该缓存它们：

```
sessions.cache()
```

执行会话分析管道是一个代价很高的操作，并且对数据建立会话之后的结果往往对我们可能要执行的多种不同分析都非常有用。如果这份数据在后续的分析中还要继续使用，或者其他数据科学家也要用到这份数据，那么可以对大规模数据进行一次性的会话分析处理，

然后把结果写入到 HDFS 上，以便用于回答一些不同的问题。这样一次性会话分析的昂贵代价就可以分摊到多个分析问题，也不失为一种不错的策略。统一的会话分析也有利于在整个数据科学小组范围内实施统一的会话定义标准，使用统一的会话标准则有助于对结果进行对等比较。

现在我们已经准备就绪，可以开始对会话数据进行分析从而得出某个区域出租车司机在卸客之后等待下一位乘车上车的平均接单等待时间了。我们将定义一个 `boroughDuration` 方法，它接受两个 `Trip` 实例作为参数，计算出第一个 `Trip` 的区域，以及第一个 `Trip` 的下车时间和第二个 `Trip` 的上车时间的 `Duration`，代码如下：

```
def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {
  val b = bLookup(t1.dropoffX, t1.dropoffY)
  val d = (t2.pickupTime - t1.dropoffTime) / 1000
  (b, d)
}
```

我们要将这个新函数应用在所有会话数据集的连续两个载客记录上。虽然这里我们可以自己写一个 `for` 循环，但也可以用 Scala Collections API 提供的 `sliding` 这一较为函数式的方法：

```
val boroughDurations: DataFrame =
  sessions.mapPartitions(trips => {
    val iter: Iterator[Seq[Trip]] = trips.sliding(2)
    val viter = iter.
      filter(_.size == 2).
      filter(p => p(0).license == p(1).license)
    viter.map(p => boroughDuration(p(0), p(1)))
  }).toDF("borough", "seconds")
```

在 `sliding` 方法的结果上调用 `filter` 保证忽略掉只有一次载客记录的会话，或者在 `license` 字段上具有不同值的任意成对的载客记录。在会话之上进行 `mapPartitions` 操作的结果是一个由键 – 值对 borough/duration 组成的 DataFrame，我们现在可以检查一下它的内容。首先应该验证大部分的等单时间是非负的：

```
boroughDurations.
  selectExpr("floor(seconds / 3600) as hours").
  groupBy("hours").
  count().
  sort("hours").
  show()
...
+-----+--------+
|hours|   count|
+-----+--------+
|   -3|       2|
|   -2|      16|
|   -1|    4253|
```

```
|    0|13359033|
|    1|  347634|
|    2|   76286|
|    3|   24812|
|    4|   10026|
|    5|    4789|
```

只有少数几条记录的等单时间为负，我们进一步仔细检查这些记录，也没发现产生这些错误数据的规律。如果从输入数据集中排除这些持续时间为负的记录，查看各区域上车时间的平均值和标准差，可以看到：

```
boroughDurations.
  where("seconds > 0 AND seconds < 60*60*4").
  groupBy("borough").
  agg(avg("seconds"), stddev("seconds")).
  show()
...
+-------------+-----------------+--------------------+
|      borough|     avg(seconds)|stddev_samp(seconds)|
+-------------+-----------------+--------------------+
|       Queens|2380.6603554494727|  2206.6572799118035|
|           NA|  2006.53571169866|  1997.0891370324784|
|     Brooklyn| 1365.394576250576|  1612.9921698951398|
|Staten Island|         2723.5625|  2395.7745475546385|
|    Manhattan| 631.8473780726746|   1042.919915477234|
|        Bronx|1975.9209786770646|   1704.006452085683|
+-------------+-----------------+--------------------+
```

数据显示曼哈顿地区的等单时间最短，为 10 分钟左右，这在我们的意料之中。布鲁克林地区的等单时间超过曼哈顿地区的两倍，乘客下车点在史丹顿岛地区的次数相对较少，司机的平均等单时间约为 45 分钟。

正如数据所示，根据乘客目的地的不同歧视性地对待乘客对出租车司机有很大的经济利益激励：如果乘客在史丹顿岛下车，司机就要空闲很长一段时间。纽约市出租车及礼车协会多年来花了很大精力来整治这种歧视性的做法，由于乘客目的地的原因而拒载的行为一经发现就要面临罚款。对乘客目的地很近的打车数据进行分析应该是比较有意思的，如果乘客的目的地很近，司机和乘客可能会发生摩擦。

8.6　小结

设想一下，我们可以把本章所用到的技术用于开发一个应用，这个应用可以根据当前的交通模式和数据中最佳候客地点的历史记录来向出租车司机建议最佳的候客地点。还可以从乘客的角度进行分析：给定当前时间、地点和天气信息，我站在街头在 5 分钟之内招呼到出租车的概率有多大？这类信息可以加入谷歌地图这类应用中，以帮助旅客确定何时出发及采用何种交通工具。

利用 Esri API 工具，可以对来自 JVM 系语言的地理空间数据进行交互式分析。这样的工具有好几个，Esri API 是其中之一，另一个是 GeoTrellis。GeoTrellis 是一个用 Scala 写的地理空间分析工具，它的设计目标之一就是易于在 Spark 中使用。第三个是基于 Java 的 GIS 工具 GeoTools。

第 9 章

基于蒙特卡罗模拟的金融风险评估

作者：桑迪·里扎

如果你想了解地质学，就研究地震。如果你想了解经济学，就研究经济萧条。

——Ben Bernanke

风险价值（Value at Risk，VaR）是一个金融统计概念，它度量在一定条件下的期望损失大小。自 1987 年美国股灾之后，VaR 迅速流行并被金融服务机构广泛使用，在金融服务机构的管理中发挥了举足轻重的作用，比如可以用它计算金融机构申请对应的信用等级所需要持有的现金量。除此之外，VaR 还可用于帮助人们更好地理解大量投资组合的风险特征，也可用于在交易执行之前提供快速决策依据。

评估 VaR 的方法极为复杂，其中许多涉及随机条件下的市场模拟，这需要进行大量计算。这些方法背后采用了一种称为**蒙特卡罗模拟**（Monte Carlo simulation）的技术。蒙特卡罗模拟中给出数千个甚至数百万个随机的市场状况，并观察这些状况对投资组合的影响。由于 Spark 本身具有高并行性，它非常适合进行蒙特卡罗模拟。Spark 可以利用数千个 CPU 核来运行随机试验并汇总结果。作为通用数据转换引擎，Spark 也擅长执行模拟前后的预处理和后处理任务。我们可以用它把原始的金融数据转换成执行模拟所需的模型参数，同样可以用它对模拟结果进行即席分析（ad-hoc analysis）。相对于传统的 HPC 环境而言，Spark 简单的编程模型使研发周期大大缩短。

现在我们给出“期望损失”的规范定义。VaR 是对投资风险的一个简单度量，它合理地估

计了一个投资组合在未来一段时间内的最大可能损失。统计量 VaR 由 3 个参数来确定：投资组合、时间跨度、*p* 值。若 *p* 值为 0.05，某投资组合未来两周的 VaR 值为 100 万美元，则表示该投资组合在两周后损失超过 100 万美元的概率为 5%。

本章还会介绍另一个相关的统计量，我们称之为**条件风险价值**（Conditional Value at Risk，CVaR），有时也叫作**期望损失**（expected shortfall）。CVaR 由巴塞尔银行业监管委员会提出，它是一个比 VaR 更好的风险度量指标。统计量 CVaR 的 3 个参数和 VaR 相同，但 CVaR 表示的是期望损失而不是截止值（cutoff value）。*p* 值为 0.05 时，某投资组合未来两周的 CVaR 值为 500 万美元，则表示该投资组合在最坏的 5% 情况下平均损失为 500 万美元。

为了对 VaR 进行建模，我们先介绍一些新的概念、方法以及工具包。具体来说，我们将介绍核密度估计、如何用 breeze-viz 工具包进行绘图、多元正态分布（multivariate normal distribution）采样和 Apache Commons Math 工具包的统计函数。

9.1 术语

本章将涉及几个金融领域的术语，现在给出它们的简单定义。

- **金融工具**
  可交易的资产，比如债券、贷款、期权或股票。金融工具在任意时刻都可以用一个值来表示，也就是资产的卖出价。

- **投资组合**
  金融机构持有的金融工具的组合。

- **回报**
  一段时间内金融工具或投资组合的价值变化。

- **损失**
  负的回报。

- **指数**
  一个假设的金融工具组合。比如纳斯达克综合指数包含了美国和世界上其他国家主要公司的约 3000 只股票和金融工具。

- **市场因素**
  给定时间点的宏观金融环境指标，比如美国的国内生产总值（GDP）指标就是一个市场因素，又如美元对欧元的汇率也是一个市场因素。我们也常把市场因素简称为**因素**。

9.2 VaR计算方法

目前为止，我们对 VaR 的定义都比较宽泛。估计该统计量需要对投资组合的工作原理及其回报的概率分布进行建模。金融机构采用了许多不同的方法计算 VaR，然而这些计算方法都来源于几种通用的方法。

9.2.1 方差–协方差法

方差－协方差（variance-covariance）是最简单的方法，其计算复杂度也最小。该模型假设每个金融工具的回报服从正态分布，这样就能估算出 VaR 值。

9.2.2 历史模拟法

历史模拟法（historical simulation）直接使用历史数据的分布推断风险值，而不依赖概要统计量。比如，为确定一个投资组合在置信水平为 95% 时的 VaR，历史数据模拟方法会参考该资产过去 100 天的市场表现并以其中表现排倒数第五的那一天的回报作为对 VaR 的估计。该方法的一个缺点是历史数据是有限的，不能包括那些没发生的假设情况。我们所用的投资组合工具的历史数据可能没有包含崩盘的情况，但我们其实也希望在这些情况下能够对投资组合的表现进行建模。已有的技术可以使历史数据模拟方法对这些问题具有稳健性，比如在数据中引入“市场冲击”，这里我们将不做介绍。

9.2.3 蒙特卡罗模拟法

本章接下来的部分将重点介绍蒙特卡罗模拟法。该方法通过模拟随机条件下的投资组合，来减少前面介绍的几种方法中的假设因素所带来的影响。当我们不能得到概率分布的解析解时，通常可以评估其概率密度函数（probability density function，PDF)，方法是对服从该概率分布的简单随机变量进行重复采样，并对采样结果进行汇总统计。更一般地，该方法：

- 定义市场条件与每个金融工具的回报之间的关系，该关系表现为拟合历史数据的模型；
- 为那些容易采样的市场条件定义分布，这些分布也拟合历史数据；
- 在随机市场条件下进行**试验**；
- 计算每次试验的投资组合总体损失，用这些损失定义损失的经验分布，即如果运行 100 次试验来估算置信水平为 95% 时的 VaR，我们会选择试验中第五大的损失值，若要计算置信水平为 95% 时的 CVaR，我们需要计算最坏的 5 次试验的平均损失。

当然，蒙特卡罗方法也不是完美的。它依赖模型来产生试验条件和推断金融工具的表现，因此这些模型必须做出简化的假设。如果这些假设不符合实际情况，那么最终得到的概率分布也不会符合实际情况。

9.3 我们的模型

蒙特卡罗风险模型通常把每个金融工具的回报分解为一组市场因素的组合。常用的市场因素包括标普 500 指数、美国 GDP 和货币汇率等。接着我们需要一个模型根据这些市场条件来预测每个金融工具的回报。我们将在模拟中使用简单的线性模型。根据之前对回报的定义，一个因素的回报为给定时间段内市场因素值的变化。举个例子，如果标普 500 指数在一段时间内从 2000 点涨到 2100 点，那么回报为 100 点。对这些因素的回报进行简单转换可以得到一组特征。也就是说，给定试验 t 的市场因素向量 m_t，通过某个转换函数 φ 得到特征向量 f_t，f_t 向量的长度可能和向量 m_t 的长度不一样。

$$f_t = \phi(m_t)$$

为每个金融工具训练一个模型，该模型给每个特征赋予一个权重。下面给出了回报的计算公式，其中 r_{it} 为试验 t 中工具 i 的回报，c_i 为金融工具 i 的截距项（intercept term），w_{ij} 为特征 j 在金融工具 i 上的回归权重，f_{tj} 为特征 j 在试验 t 中产生的随机值：

$$r_{it} = c_i + \sum_{j=1}^{|w_i|} w_{ij} * f_{tj}$$

上述公式表示，每个金融工具的回报等于所有市场因素特征的回报与金融工具的权重的乘积之和。我们可以用历史数据来拟合每个金融工具的线型模型（也称为线性回归）。如果 VaR 的时间跨度为两周，回归问题把每个间隔两周的时间点当作具有标号的样本点。

需要指出的是，我们也可选用更复杂的模型。比如可以不用线性模型，而是用回归树技术或在模型中显式地加入特定领域的知识。

有了从市场因素中计算金融工具损失的模型之后，还需要一个模拟市场因素的方法。我们简单假设市场因素回报服从正态分布。为了考虑市场因素之间的相关性（比如纳斯达克指数下跌时道琼斯指数也很可能跟着下跌），我们使用多元正态分布，其协方差矩阵是非对角阵：

$$m_t \sim \mathcal{N}(\mu, \Sigma)$$

这里 μ 代表因素回报经验平均向量，Σ 代表市场因素回报经验的协方差矩阵。

与前面讨论的一样，在模拟市场因素时，我们也可以选择更加复杂的方法。可以假定每个市场因素服从不同的分布类型，比如采用尾部更厚的分布。

9.4 获取数据

要找到大量格式规整的历史价格数据并非易事，但我们可以在 Yahoo! 上下载到大量 CSV 格式的股票数据。下面的脚本使用一系列 REST 调用来下载纳斯达克指数里所有股票的历

史数据，并将其存放在 stocks/ 目录下。该脚本在本书 GitHub 资料库的 risk/data 目录下：

```
$ ./download-all-symbols.sh
```

我们也需要这份历史数据的风险因素，包括标普 500 和纳斯达克指数值，还有 5 年期以及 30 年期国债价格数据。标普 500 和纳斯达克指数数据同样可以从 Yahoo! 下载：

```
$ mkdir factors/
$ ./download-symbol.sh ^GSPC factors
$ ./download-symbol.sh ^IXIC factors
$ ./download-symbol.sh ^TYX factors
$ ./download-symbol.sh ^FVX factors
```

9.5 数据预处理

从 Yahoo! 获取的 GOOGL 股票数据的前几行如下：

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-10-24,554.98,555.00,545.16,548.90,2175400,548.90
2014-10-23,548.28,557.40,545.50,553.65,2151300,553.65
2014-10-22,541.05,550.76,540.23,542.69,2973700,542.69
2014-10-21,537.27,538.77,530.20,538.03,2459500,538.03
2014-10-20,520.45,533.16,519.14,532.38,2748200,532.38
```

接下来启动 Spark shell。在本章中会使用几个库来简化工作。GitHub 仓库中有一个 Maven 项目，用来将所有这些依赖关系打包成一个 JAR 文件：

```
$ cd ch09-risk/
$ mvn package
$ cd data/
$ spark-shell --jars ../target/ch09-risk-2.0.0-jar-with-dependencies.jar
```

对每个数据源的每个金融工具和市场因素，我们可以用 `(date, closing price)` 元组列表来描述。`java.time` 库提供表示和操作日期的实用功能。我们可以日期表示为 `LocalDate` 对象，还可以使用 `DateTimeFormatter` 来解析 Yahoo! 日期格式：

```
import java.time.LocalDate
import java.time.format.DateTimeFormatter

val format = DateTimeFormatter.ofPattern("yyyy-MM-dd")
LocalDate.parse("2014-10-24")
res0: java.time.LocalDate = 2014-10-24
```

3000 个金融工具加 4 个市场因素的历史数据量较小，可以在本地进行读取和处理。即使对于涉及几十万个金融工具和几千个市场因素的较大型的模拟来说也是如此。然而，当模拟真正运行起来时，每个工具都需要进行大量的计算，这时我们就需要 Spark 这类分布式系统了。

现在读取全部的 Yahoo! 历史数据，代码如下：

```
import java.io.File

def readYahooHistory(file: File): Array[(LocalDate, Double)] = {
  val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")
  val lines = scala.io.Source.fromFile(file).getLines().toSeq
  lines.tail.map { line =>
    val cols = line.split(',')
    val date = LocalDate.parse(cols(0), formatter)
    val value = cols(1).toDouble
    (date, value)
  }.reverse.toArray
}
```

注意 `lines.tail` 用于去掉标题行。现在我们加载所有数据并过滤掉历史数据不足 5 年的金融工具，代码如下：

```
val start = LocalDate.of(2009, 10, 23)
val end = LocalDate.of(2014, 10, 23)

val stocksDir = new File("stocks/")
val files = stocksDir.listFiles()
val allStocks = files.iterator.flatMap { file =>> ❶
  try {
    Some(readYahooHistory(file))
  } catch {
    case e: Exception => None
  }
}
val rawStocks = allStocks.filter(_.size >= 260 * 5 + 10)

val factorsPrefix = "factors/"
val rawFactors = Array(
  "^GSPC.csv", "^IXIC.csv", "^TYX.csv", "^FVX.csv").
  map(x => new File(factorsPrefix + x)).
  map(readYahooHistory)
```

❶ 这里使用迭代器来对文件做流式处理，不用一次性把全部内容加载到内存。

由于不同金融工具的交易日期可能不相同，或者由于其他原因数据中有些值缺失，因此我们有必要对不同的历史数据进行规范化处理。首先需要将时间序列数据统一到同一个时间区间。然后对有缺失的数据，需要为其填充数据。对于时间序列数据中缺失开始时间和结束时间的情况，我们用附近的日期填充即可，代码如下：

```
def trimToRegion(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate)
  : Array[(LocalDate, Double)] = {
  var trimmed = history.dropWhile(_._1.isBefore(start)).
    takeWhile(x => x._1.isBefore(end) || x._1.isEqual(end))
  if (trimmed.head._1 != start) {
    trimmed = Array((start, trimmed.head._2)) ++ trimmed
  }
```

```
  if (trimmed.last._1 != end) {
    trimmed = trimmed ++ Array((end, trimmed.last._2))
  }
  trimmed
}
```

对于一个时间序列的数据存在缺失值的情况，我们使用该工具上一个交易日的收盘价来代替。因为 Scala 集合并没有提供现成的方法帮我们完成这个任务，所以我们还得自己写。spark-ts 库（https://github.com/sryza/spark-timeseries）和 flint 库（https://github.com/twosigma/flint）是替代方案，它们也有许多实用的时间序列处理函数。

```
import scala.collection.mutable.ArrayBuffer

def fillInHistory(history: Array[(DateTime, Double)],
    start: DateTime, end: DateTime): Array[(DateTime, Double)] = {
  var cur = history
  val filled = new ArrayBuffer[(DateTime, Double)]()
  var curDate = start
  while (curDate < end) {
    if (cur.tail.nonEmpty && cur.tail.head._1 == curDate) {
      cur = cur.tail
    }

    filled += ((curDate, cur.head._2))

    curDate += 1.days
    // 跳过周末
    if (curDate.dayOfWeek().get > 5) curDate += 2.days
  }
  filled.toArray
}
```

将 `trimToRegion` 和 `fillInHistory` 函数应用在数据上：

```
val stocks = rawStocks.
  map(trimToRegion(_, start, end)).
  map(fillInHistory(_, start, end))

val factors = (factors1 ++ factors2).
  map(trimToRegion(_, start, end)).
  map(fillInHistory(_, start, end))
```

请记住，即使这里使用的 Scala API 与 Spark API 非常相似，这些操作也是在本地执行的。`stocks` 的每个元素都是由某支股票在不同时间点的价格组成的数组。`factors` 结果和 `stocks` 一样。这些数组的长度应该都相等，我们可以用如下代码进行验证：

```
(stocks ++ factors).forall(_.size == stocks(0).size)
res17: Boolean = true
```

9.6 确定市场因素的权重

回顾一下，VaR 值代表一个给定时间段内的可能损失大小。我们关心的不是金融工具的绝对价格，而是在**一段时间内**金融工具价格的变化。在本章的计算中，我们将时间跨度设为两周。下面的函数利用了 Scala 集合的 `sliding` 方法将价格的时间序列转换成间隔为 2 周的价格移动交叠序列。注意，由于金融数据中不考虑周末，所以时间窗口为 10 而不是 14：

```
def twoWeekReturns(history: Array[(LocalDate, Double)])
  : Array[Double] = {
  history.sliding(10).
    map { window =>
      val next = window.last._2
      val prev = window.head._2
      (next - prev) / prev
    }.toArray
}

val stocksReturns = stocks.map(twoWeekReturns).toArray.toSeq ❶
val factorsReturns = factors.map(twoWeekReturns)
```

❶ 由于我们之前使用了迭代器，`stocks` 是一个迭代器。`.toArray.toSeq` 遍历 `stocks`，并将内存中的元素收集到一个序列中。

有了回报的历史数据，我们就可以回过来看看如何训练模型来预测金融工具回报。我们希望有一个模型可以根据两周内市场因素的回报来预测每个金融工具在相同时间段内的回报。为了简化问题，我们使用线性回归模型。

金融工具的回报与市场因素之间可能是非线性关系，为了对这个情况进行建模，我们可以在模型中加入一些附加的特征。对市场因素回报进行非线性变换可以得到这些特征。这里我们尝试对每个市场因素增加两个附加特征：市场因素的平方以及平方根。由于应变量仍然是特征的线性函数，从这个意义上讲，我们的模型仍然是线性模型，只不过有些特征正好由市场因素的非线性函数确定而已。请记住我们这里采用的这种特征转换只是为了说明问题，而在金融建模实践中，进行预测时采用的做法可能并不相同。

虽然由于每个金融工具都对应一次回归，我们这里执行了许多次回归，但是特征的数量和每次回归的数据量是其实是很小的。因此，在建立线性模型的过程中我们没必要使用 Spark 进行分布式运算，只要用 Apache Commons Math 工具包提供的普通最小二乘回归功能就足够了。虽然现在我们的市场因素数据是由历史数据组成的 `Seq`（每个 `Seq` 都是由 `(DateTime, Double)` 二元组组成的数组），但 `OLSMultipleLinearRegression` 要求数据为样本点数组（对我们的示例来说就是 2 周的时间段），所以我们需要对市场因素矩阵进行变换，代码如下：

```
def factorMatrix(histories: Seq[Array[Double]])
  : Array[Array[Double]] = {
  val mat = new Array[Array[Double]](histories.head.length)
```

```
  for (i <- histories.head.indices) {
    mat(i) = histories.map(_(i)).toArray
  }
  mat
}

val factorMat = factorMatrix(factorsReturns)
```

现在我们可以处理附加的特征了，代码如下：

```
def featurize(factorReturns: Array[Double]): Array[Double] = {
  val squaredReturns = factorReturns.
    map(x => math.signum(x) * x * x)
  val squareRootedReturns = factorReturns.
    map(x => math.signum(x) * math.sqrt(math.abs(x)))
  squaredReturns ++ squareRootedReturns ++ factorReturns
}

val factorFeatures = factorMat.map(featurize)
```

然后我们可以拟合线性模型。另外，为了找到每个工具的模型参数，我们可以使用 `OLSMultipleLinearRegression` 的 `estimateRegressionParameters` 方法：

```
import org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression

def linearModel(instrument: Array[Double],
    factorMatrix: Array[Array[Double]])
  : OLSMultipleLinearRegression = {
  val regression = new OLSMultipleLinearRegression()
  regression.newSampleData(instrument, factorMatrix)
  regression
}

val factorWeights = stocksReturns.
  map(linearModel(_, factorFeatures)).
  map(_.estimateRegressionParameters()).
  toArray
```

现在我们得到了一个 1867×8 的矩阵，其中每一行代表一个金融工具的模型参数集合（这些参数可能是系数、权重、协变量、回归因子等）。

为了节省篇幅，我们省略了分析过程，但在这个点上，对于一个实际的应用处理管道（pipeline），有必要了解模型对数据的拟合程度。因为数据点是从时间序列上得到的，特别是时间窗口是交叠的，所以这些样本很有可能是自相关的（autocorrelated）。这就是说，如果采用像 R^2 之类的度量，我们很可能对模型的拟合程度做出乐观估计。Breusch-Godfrey 测试（http://en.wikipedia.org/wiki/Breusch–Godfrey_test）是对自相关性的影响进行评估的一种标准测试。这种快速评估模型的方法就是将时间序列拆分成两个集合。拆分时要注意取出的点处于中间位置，数据点要足够多，要保证前面一组的后面的点与后面一组的前面的点不是自相关的。然后在这个集合上进行模型训练，在另一个集合上检验误差。

9.7 采样

有了将市场因素回报映射到金融工具回报的模型之后，接下来可以讨论怎样生成随机回报因素来模拟市场条件。也就是说，我们需要确定因素回报向量的一个概率分布，并从该分布上采样。数据实际服从什么分布呢？为了回答此类问题，有必要先对数据进行可视化。连续概率分布的可视化可以采用密度曲线，它给出了在分布区间上的概率密度函数（PDF）。因为我们不知道数据服从的分布，所以并没有一个公式可以帮助我们计算任意点上的概率密度。但我们可以使用一种称为**核密度估计**（kernel density estimation）的技术来粗略估计概率密度。不严格地讲，核密度估计是一种对直方图进行平滑处理的方法。它以每个数据点为中心建立一个概率分布（通常为正态分布），因此一个两周回报样本的集合将有 200 个正态分布，每个分布的总体均值都不一样。为了评估在给定点的概率密度，可以计算所有正态分布在这个点上的概率密度，然后取平均值。核密度曲线的平滑程度取决于它的**带宽**（bandwidth），也就是每个正态分布的标准差。本书的 GitHub 资料库上提供了一个核密度估计的实现，既可以用于 RDD，也可用于本地集合。为了节省篇幅，这里不再赘述。

breeze-viz 是一个 Scala 工具，我们可以用它轻松地绘制简单图形。下面的代码绘制了样本集的密度曲线：

```
import org.apache.spark.mllib.stat.KernelDensity
import org.apache.spark.util.StatCounter
import breeze.plot._

def plotDistribution(samples: Array[Double]): Figure = {
  val min = samples.min
  val max = samples.max
  val stddev = new StatCounter(samples).stdev
  val bandwidth = 1.06 * stddev * math.pow(samples.size, -0.2) ❶

  val domain = Range.Double(min, max, (max - min) / 100).
    toList.toArray
  val kd = new KernelDensity().
    setSample(samples.toSeq.toDS.rdd).
    setBandwidth(bandwidth)
  val densities = kd.estimate(domain)
  val f = Figure()
  val p = f.subplot(0)
  p += plot(domain, densities)
  p.xlabel = "Two Week Return ($)"
  p.ylabel = "Density"
  f
}
```

❶ 我们使用众所周知的“西尔弗曼的经验法则”（Silverman’s rule of thumb），它以英国统计学家伯纳德·西尔弗曼（Bernard Silverman）的名字命名，用于选择合理的带宽。

图 9-1 显示了标普 500 历史价格两周回报的概率分布（概率密度函数）。

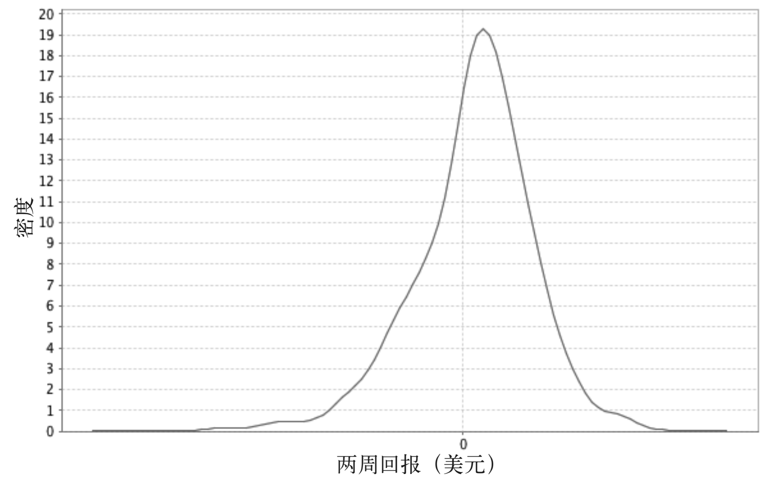


图 9-1：标普 500 两周回报分布

图 9-2 显示了 30 年期国债两周回报的概率分布。

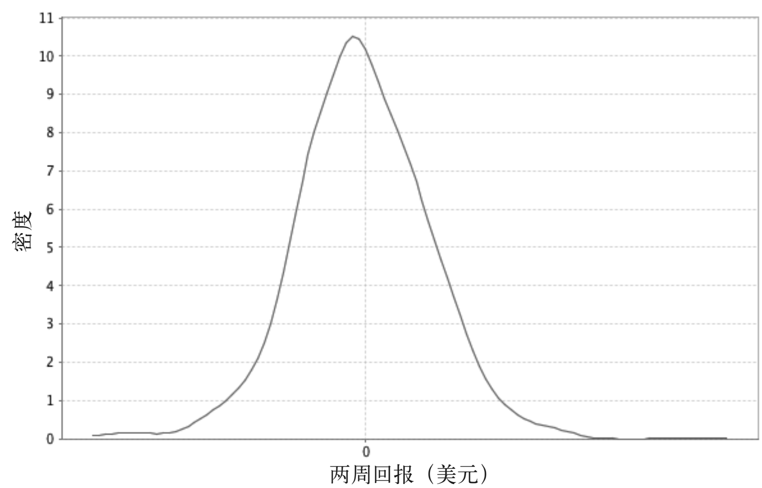


图 9-2：30 年期国债两周回报分布

我们将为每个因素回报拟合一个正态分布。有时候值得多花些精力寻找一个更符合实际情况的分布，比如尾部更厚的分布，能更好地拟合数据。但这里为了节省篇幅，就不深入介绍模拟的调优方法了。

最简单的因素回报采样方法是用一个正态分布拟合每个因素，然后对这个分布进行独立采样。然而，这里我们忽略了市场因素常常是相关的这个实际情况。如果标普指数下跌，道琼斯指数也可能跟着下跌。如果没有考虑到这些相关性，我们得到的风险预测（risk profile）的噪声将比实际情况要大。我们的市场因素是否相关呢？ Commons Math 中的皮尔森相关系数实现可以帮我们回答这个问题，代码如下：

```
import org.apache.commons.math3.stat.correlation.PearsonsCorrelation

val factorCor =
  new PearsonsCorrelation(factorMat).getCorrelationMatrix().getData()
println(factorCor.map(_.mkString("\t")).mkString("\n"))
1.0       -0.3472   0.4424    0.4633 ❶
-0.3472   1.0       -0.4777   -0.5096
0.4424    -0.4777   1.0       0.9199
0.4633    -0.5096   0.9199    1.0
```

❶ 为了统一格式，我们只保留了小数点后的部分位数。

由于非对角线上有非零值，所以看来市场因素之间存在相关性。

多元正态分布

要考虑因素之间的相关性，可以使用多元正态分布。多元正态分布的每个样本是一个向量，在其他所有维度的值都确定的情况下，对于给定维度的值服从正态分布。但是，多个变量的联合分布并不是独立分布。

多元正态分布的参数为对应每个维度上的均值向量和一个矩阵，该矩阵描述了任意两个维度之间的协方差。对于 N 维的情况，由于我们要得到任何两个维度之间的协方差，所以协方差矩阵为 N 乘 N。如果协方差矩阵为对角阵，多元正态分布就退化成独立分布，但如果非对角线上存在非零值，那么表示相应的两个变量之间存在相关性。

VaR 相关文献常常提到因素权重转换步骤，经过权重转换，因素之间的相关性被去掉了，这样就可以进行采样了。这里常常用到楚列斯基分解（Cholesky decomposition）或特征分解（eigendecomposition）。这里我们可以直接调用 Apache Commons Math 工具包的 `MultivariateNormalDistribution`，它在底层使用了特征分解。

为了在本章数据上拟合一个多元正态分布，我们首先要得到其样本均值和协方差：

```
import org.apache.commons.math3.stat.correlation.Covariance

val factorCov = new Covariance(factorMat).getCovarianceMatrix().
  getData()

val factorMeans = factorsReturns.
  map(factor => factor.sum / factor.size).toArray
```

接下来，以上面得到的均值和协方差为参数创建一个分布：

```
import org.apache.commons.math3.distribution.MultivariateNormalDistribution

val factorsDist = new MultivariateNormalDistribution(factorMeans,
  factorCov)
```

从分布中对市场条件进行一系列采样：

```
factorsDist.sample()
res1: Array[Double] = Array(-0.05782773255967754, 0.01890770078427768,
  0.029344325473062878, 0.04398266164298203)

factorsDist.sample()
res2: Array[Double] = Array(-0.009840154244155741, -0.01573733572551166,
  0.029140934507992572, 0.028227818241305904)
```

9.8 运行试验

讨论完每个金融工具的模型和市场因素回报的采样过程，现在就可以开始运行实际的试验了。由于运行试验是个计算密集型的任务，所以我们最终还是要用 Spark 来对其并行化。在每次试验中，我们希望提取一组风险因素样本，用该样本预测每个金融工具的回报，然后将所有回报相加得到总体试验损失。为了使分布具有代表性，我们需要运行数千次甚至是数百万次试验。

有几种方式可以对模拟进行并行化，比如可以对试验进行并行化，也可以对金融工具进行并行化，或者同时对二者进行并行化。如果同时进行并行化，我们要创建一个金融工具的数据集和一个试验参数的数据集，然后用笛卡儿转换 `cartesian` 来生成一个包含所有组合的数据集。这种方法最通用，但有两个缺点：第一，该方法需要显式地创建试验参数 RDD，而这其实可以通过设置随机种子来避免；第二，需要进行乱序操作。

对金融工具进行并行化的代码如下：

```
val randomSeed = 1496
val instrumentsDS = ...
def trialLossesForInstrument(seed: Long, instrument: Array[Double])
  : Array[(Int, Double)] = {
  ...
}
instrumentsDS.flatMap(trialLossesForInstrument(randomSeed, _)).
  reduceByKey(_ + _)
```

采用这种方式时，数据按照金融工具对 RDD 进行分区，对每个金融工具进行 `flatMap` 转换就可以得到每次试验的损失。对所有任务采用相同随机种子意味着生成的试验序列是相同的。`reduceByKey` 操作把同一个试验的对应的所有损失都汇总在一起。这种方式的缺点是它也需要进行乱序，数据量量级为 `O(|instruments| * |trials|)`。

本章中的几千个金融工具的模型数据非常小，所以可以直接放入每个执行器（executor）的内存里。粗略估算一下，即使有 100 万个工具和数百个因素，执行器的内存也能存下。100 万个工具乘以 500 个因素，再乘以每个因素权重所需的 8 字节，总共约 4 GB，对当今大多数集群机器而言，将这些数据存放到每个执行器的内存里是完全可行的。因此我们应该将金融工具数据设为广播变量，每个执行器都有完整的金融工具数据的好处在于，每次实验的总体损失在单台机器上就能算出，这样就没必要在机器之间进行汇总。

对于按实验进行分区的方法（我们将使用这种方法），首先需要生成一个随机种子组成的 RDD，我们希望每个分区的随机种子都不一样，这样每个分区将产生不同的实验，代码如下：

```
val parallelism = 1000
val baseSeed = 1496

val seeds = (baseSeed until baseSeed + parallelism)
val seedDS = seeds.toDS().repartition(parallelism)
```

随机数生成是一个耗时的过程，也是 CPU 密集型的任务。预先生成一组随机数然后在多个作业中使用通常效率较高，但本章不使用这个方法。由于蒙特卡罗实验假定随机数服从独立分发，因此为了符合该假设，**不能**在同一个作业中使用相同的随机数。如果要采用事先生成随机数的方法，我们只需将代码中的 `toDS` 方法替换为 `textFile`，并加载事先生成好的 `randomNumbersDS` 文件即可。

对每个随机种子，我们希望生成一组实验参数并观察这些参数对所有金融工具的影响。我们从底层开始，先写一个函数计算单个实验中单个工具的回报，只需应用之前训练好的每个工具对应的线性模型即可。由于回归参数的 `instrument` 数组包含了截距（intercept term），所以它的长度比 `trial` 数组大 1：

```
def instrumentTrialReturn(instrument: Array[Double],
    trial: Array[Double]): Double = {
  var instrumentTrialReturn = instrument(0)
  var i = 0
  while (i < trial.length) { ❶
    instrumentTrialReturn += trial(i) * instrument(i+1)
    i += 1
  }
  instrumentTrialReturn
}
```

❶ 因为此处对性能有很大影响，所以使用 `while` 循环，而没有用 Scala 的函数式编程。

接着，只需将所有工具的回报相加，即可得到单个实验的全体回报。这假定我们持有的投资组合中每个工具的价值相同。如果每只股票持有的数量不同，则使用加权平均。

```
def trialReturn(trial: Array[Double],
    instruments: Seq[Array[Double]]): Double = {
  var totalReturn = 0.0
  for (instrument <- instruments) {
    totalReturn += instrumentTrialReturn(instrument, trial)
  }
  totalReturn / instruments.size
}
```

最后，我们需要在每个任务中生成一系列实验。由于随机数的选择占该过程的很大一部分，所以有必要选用更强大的随机数生成器，这样就不容易产生重复的随机数。Commons Math 包中 Mersenne twister 的实现很适合，根据前面提到的方法，我们使用它对多元正态分布进行采样。注意，为了将生成的因素回报转换成模型中所需的特征格式，我们在生成的因素回报上应用了刚定义的 `featurize` 方法：

```
import org.apache.commons.math3.random.MersenneTwister

def trialReturns(seed: Long, numTrials: Int,
    instruments: Seq[Array[Double]], factorMeans: Array[Double],
    factorCovariances: Array[Array[Double]]): Seq[Double] = {
  val rand = new MersenneTwister(seed)
  val multivariateNormal = new MultivariateNormalDistribution(
    rand, factorMeans, factorCovariances)

  val trialReturns = new Array[Double](numTrials)
  for (i <- 0 until numTrials) {
    val trialFactorReturns = multivariateNormal.sample()
    val trialFeatures = featurize(trialFactorReturns)
    trialReturns(i) = trialReturn(trialFeatures, instruments)
  }
  trialReturns
}
```

现在准备工作就绪，我们可以用上述函数计算出一个 RDD，这个 RDD 的每个元素代表一次实验的总体回报。由于金融工具数据（即每个因素对每个工具的权重矩阵）很大，我们将它设为广播变量。这样每个执行器都只要对它进行一次反序列化即可。

```
val numTrials = 10000000

val trials = seedDS.flatMap(
  trialReturns(_, numTrials / parallelism,
    factorWeights, factorMeans, factorCov))

trials.cache()
```

现在回顾一下，我们对这些数字所做的所有操作都是为了计算 VaR。`trials` 现在代表了投资组合回报的经验分布。要计算置信水平为 95% 时的 VaR，需要找到在最差的 5% 和最好的 5% 的情况下的回报。有了经验分布，要得到这两个回报，只要找到经验分布上的一个实验，对于该实验，有 5% 的实验回报比它低，并且有 95% 的实验的回报比它高。在驱动

程序中，用 takeOrdered 行动从所有实验中取出最差的 5%，就可以达到这个目的。这个表现最差的实验回报集合中的最高回报即为我们要求的 VaR。

```
def fivePercentVaR(trials: Dataset[Double]): Double = {
  val quantiles = trials.stat.approxQuantile("value",
    Array(0.05), 0.0)
  quantiles.head
}

val valueAtRisk = fivePercentVaR(trials)
valueAtRisk: Double = -0.010831826593164014
```

用几乎完全一样的方法也能求出 CVaR。不过求 CVaR 时我们取最差的 5% 的实验回报集合的平均回报，而不是其中的最高回报。

```
def fivePercentCVaR(trials: Dataset[Double]): Double = {
  val topLosses = trials.orderBy("value").
    limit(math.max(trials.count().toInt / 20, 1))
  topLosses.agg("value" -> "avg").first()(0).asInstanceOf[Double]
}

val conditionalValueAtRisk = fivePercentCVaR(trials)
conditionalValueAtRisk: Double = -0.09002629251426077
```

9.9 回报分布的可视化

除了计算一定置信水平下的 VaR 之外，我们还可以用它更全面地了解回归分布。回报服从正态分布吗？在极端情况下回报会不稳定吗？与我们之前为每个市场因素做过的方法类似，我们可以用核密度估计来估算联合概率分布的概率密度函数（见图 9-3）。再次说明，分布式估算核密度的代码（采用 RDD）可以参考本书附带的 GitHub 资料库：

```
import org.apache.spark.sql.functions

def plotDistribution(samples: Dataset[Double]): Figure = {
  val (min, max, count, stddev) = samples.agg(
    functions.min($"value"),
    functions.max($"value"),
    functions.count($"value"),
    functions.stddev_pop($"value")
  ).as[(Double, Double, Long, Double)].first()
  val bandwidth = 1.06 * stddev * math.pow(count, -0.2) ❶

  // 在toArray之前使用toList是为了避开Scala的一个bug
  val domain = Range.Double(min, max, (max - min) / 100).
    toList.toArray
  val kd = new KernelDensity().
    setSample(samples.rdd).
    setBandwidth(bandwidth)
  val densities = kd.estimate(domain)
```

```
    val f = Figure()
    val p = f.subplot(0)
    p += plot(domain, densities)
    p.xlabel = "Two Week Return ($)"
    p.ylabel = "Density"
    f
  }

plotDistribution(trials)
```

❶ 又是西尔弗曼的经验法则。

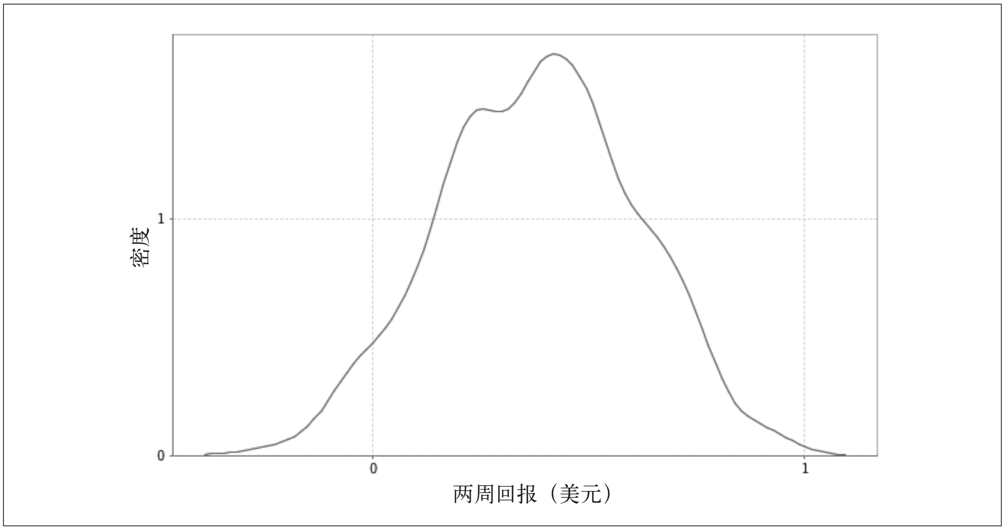


图 9-3：两周回报分布

9.10 结果的评估

怎样才能确定风险评估的好坏？怎样才能知道是否该进行更多次实验？一般来说，蒙特卡罗模拟的误差与$1/\sqrt{n}$成正比。也就是说，通常实验次数增加 4 倍，那么误差就大约减半。

自举算法是计算 VaR 统计量置信区间的好方法。通过在实验得到的投资组合回报集合上进行重复放回采样（repeatedly sampling with replacement），可以得到一个 VaR 的自举分布。每次我们从样本中取出与实验次数相等的样本，并根据这些取出的样本计算 VaR。所有计算出的 VaR 就形成了一个经验分布，只要根据该经验分布的分位数就能得到置信区间了。

下面给出计算 RDD 的所有统计量（即函数的 `computeStatistic` 参数）的自举置信区间。注意该函数使用了 Spark 的 `sample` 方法，第一个参数 `withReplacement` 设为 `true`，第二个参数设为 1.0，代表采样大小等于数据集大小：

```
def bootstrappedConfidenceInterval(
    trials: Dataset[Double],
    computeStatistic: Dataset[Double] => Double,
    numResamples: Int,
    probability: Double): (Double, Double) = {
  val stats = (0 until numResamples).map { i =>
    val resample = trials.sample(true, 1.0)
    computeStatistic(resample)
  }.sorted
  val lowerIndex = (numResamples * probability / 2 - 1).toInt
  val upperIndex = math.ceil(numResamples * (1 - probability / 2))
    .toInt
  (stats(lowerIndex), stats(upperIndex))
}
```

接下来我们调用这个函数，并传入前面定义好的 `fivePercentVaR` 函数以从实验 RDD 中计算 VaR：

```
bootstrappedConfidenceInterval(trials, fivePercentVaR, 100, .05)
...
(-0.019480970253736192,-1.4971191125093586E-4)
```

同样我们可以计算自举 CVaR：

```
bootstrappedConfidenceInterval(trials, fivePercentCVaR, 100, .05)
...
(-0.10051267317397554,-0.08058996149775266)
```

置信区间提供了模型对于结果的置信水平信息，但并没有提供模型是否符合实际情况的信息。对于检测结果质量，在历史数据上进行回测（backtesting）是个不错的方法。对 VaR 的测试通常采用 Kupiec 提出的失败频率检验法（Proportion-of-Failures，POF）。POF 计算在多个历史时间段内的投资组合回报，然后计算损失超过 VaR 的次数。备择假设认为 VaR 是合理的，充分极限检验统计量认为 VaR 没有准确估计数据。下面我们给出检验统计量的公式，它依赖如下 3 个参数：计算 VaR 的置信水平参数 p、损失超过 VaR 的历史时间段的次数 x 和历史时间段的总次数 T：

$$-2\ln\left(\frac{(1-p)^{T-x}p^{x}}{\left(1-\frac{x}{T}\right)^{T-x}\left(\frac{x}{T}\right)^{x}}\right)$$

下面是在历史数据上计算检验统计量的代码。为了数值计算的稳定性更好，我们放大了对数值。

$$-2\left((T-x)\ln(1-p)+x\ln(p)-(T-x)\ln\left(1-\frac{x}{T}\right)-x\ln\left(\frac{x}{T}\right)\right)$$

```
var failures = 0
for (i <- stocksReturns.head.indices) {
```

```
    val loss = stocksReturns.map(_(i)).sum / stocksReturns.size
    if (loss < valueAtRisk) {
      failures += 1
    }
  }
  failures
  ...
  257

  val total = stocksReturns.size
  val confidenceLevel = 0.05
  val failureRatio = failures.toDouble / total
  val logNumer = ((total - failures) * math.log1p(-confidenceLevel) +
    failures * math.log(confidenceLevel))
  val logDenom = ((total - failures) * math.log1p(-failureRatio) +
    failures * math.log(failureRatio))
  val testStatistic = -2 * (logNumer - logDenom)
  ...
  180.3543986286574
```

如果备择假设为 VaR 是合理的，那么该检验统计量服从自由度为 1 的卡方分布。我们可以用 Commons Math 类 `ChiSquaredDistribution` 来计算检验统计值对应的 p 值：

```
import org.apache.commons.math3.distribution.ChiSquaredDistribution

1 - new ChiSquaredDistribution(1.0).cumulativeProbability(testStatistic)
```

结果 p 值很小，它表示我们有充足的证据拒绝"模型是合理的"这个零假设。虽然我们之前得到了相当紧密的置信区间，表明我们的模型在内部是一致的，但是测试结果表明，它与观察的实际情况并不相符。看来我们还需要进一步改进我们的模型。

9.11 小结

相对于金融机构实际应用的模型来说，本章练习中构造的模型还是一个非常粗略的初步结果。要构造一个准确的 VaR 模型，还有一些非常重要的步骤，但本章只进行了粗略的讨论。比如，市场因素的选择决定了模型的好坏，金融机构常常在模拟中引入数百个市场因素。

选择这些因素不但需要在历史数据上运行无数次试验，而且需要大量创新性实践。将市场因素映射为工具回报的预测模型也相当重要。本章中我们用了简单的线性模型，但许多模拟采用非线性函数或模拟布朗运动的时间轨迹。最后，还应该注意用于模拟因素回报的分布函数，Kolmogorov-Smirnoff 测试和卡方测试常用于测试经验分布的正态性，Q-Q 曲线图可以形象地比较不同分布。相比本章中采用的正态分布，尾部更厚的分布曲线通常能更好地反映金融风险。要想更好地了解此类分布曲线，可以参考 Markus Haas 和 Christian Pigorsch 的文章"Financial Economics, Fat-tailed Distributions"。

银行也使用 Spark 和大规模数据处理框架基于历史数据计算 VaR。想了解基于历史数据的 VaR 计算方法的概况和不同方法的表现，可以参考 Darryll Hendricks 的论文“Evaluation of Value-at-Risk Models Using Historical Data”（https://nyfed.org/1ACaI2O）。

蒙特卡罗风险模拟的作用并不只是计算单个统计量。通过影响投资决策，其结果还可用于主动降低投资组合的风险。举例来说，如果在回报最差的实验中一个特定的工具集合常常多次造成损失，就可以考虑将这些工具从投资组合中去掉，或者增加逆向对冲工具。

第10章

基因数据分析和BDG项目

作者：于里·莱瑟森

我们需要把地球上的 SCHPON（硫、碳、氢、磷、氧和氮，各种“卵”）发射到外太空。

——George M. Church

随着下一代 DNA 测序（next-generation DNA sequencing，NGS）技术的出现，生命科学迅速变成了一个数据驱动的领域。然而如何充分利用这些数据，对传统计算生态系统是个不小的挑战。这些传统系统的分布式计算基于底层操作原语（比如 DRMAA 或 MPI）构造，所以它们很难用，而且使用纷繁复杂的半结构文本格式。

本章主要有 3 个目标。第一，面向普通 Spark 用户介绍一些新的序列化和文件格式（Avro 和 Parquet），这些格式可以很好地与 Hadoop 结合，大大简化了数据管理的许多问题。使用这些序列化技术可以实现紧凑的二进制表示、面向服务的架构和跨语言的兼容性，对许多情况我们都推荐使用它们。第二，面向那些有经验的生物信息学家介绍在 Spark 中如何完成典型的基因学任务。

具体来说，我们用 Spark 操作大量基因学数据，对其进行处理、过滤，构造转录因子结合位点预测模型，并把 ENCODE（https://www.encodeproject.org/）基因组标注与 1000 个 Genome 项目变体（http://www.internationalgenome.org/）进行联结。最后，本章还可作为 ADAM 项目的教程。ADAM 项目提供了一组基因学相关的 Avro 模式，以及大规模基因学分析的 Spark API 和命令行工具。ADAM 项目还基于 Hadoop 和 Spark 提供了 GATK

（Genome Analysis Toolkit，https://software.broadinstitute.org/gatk/best-practices/）最佳实践的原生分布式实现。

本章介绍基因学的部分面向有经验的生物信息学家，他们对其中的典型问题比较熟悉。但数据序列化部分对任何要处理大量数据的读者都适用。新手如果感兴趣的话，生物学入门知识可以参考 Eric Lander 的 EdX 课程（https://www.edx.org/course/introduction-biology-secret-life-mitx-7-00x-6）。有关生物信息学的介绍，请参阅 Arthur Lesk 的著作 *Introduction to Bioinformatics*。

最后，因为基因组隐含着一个一维坐标系统，所以许多基因组操作本质上是空间操作。ADAM 项目提供了针对基因组学的 API，以及使用旧版 RDD 接口执行分布式空间连接的实现。因此，本章继续使用原来的接口，而不是基于 Dataset 和 DataFrame 的新接口。

RDD 使用说明

与本书其余章节不同，本章和下一章将利用 Spark 旧版的 RDD API。主要原因是 ADAM 项目已经实现了许多专门用于一维几何计算的连接算子，这些计算在基因组学处理中是很常见的。这些操作还没有移植到新的 Dataset API 上，不过移植已经在路线图上了。此外，DataFrame API 抽象掉了更多分布式计算的细节，但移植 ADAM 连接运算符需要和 Spark 的查询计划器交互。另一方面，当读者遇到使用 RDD API 的场景时，比如使用其他 Spark 库或遗留代码，本章也可以作为参考资料。

10.1 分离存储与模型

生物信息学家在数据格式上花了太多精力，我们简单罗列一下这些格式，包括 .fasta、.fastq、.sam、.bam、.vcf、.gvcf、.bcf、.bed、.gff、.gtf、.narrowPeak、.wig、.bigWig、.bigBed、.ped、.tped 等一大串。更不用说这些科学家花了多少精力研究每个定制工具的定制格式。然而这些格式的规范要么不完整，要么太模糊（这样很难保证实现的一致性或兼容性），而且它们使用 ASCII 编码数据。ASCII 数据在生物信息学中用得非常普遍，但编码效率不高，压缩比相对较差。社区已经开始改进这些规范的不足之处，参见 https://github.com/samtools/hts-specs。除此之外，数据必须经过解析及其他计算处理。因为实质上这些格式只有几种常用对象类型：对齐序列读数（aligned sequence read）、命名基因型（called genotype）、序列特征和表现型（phenotype），所以尤其麻烦。（术语“序列特征”在基因学中的含义有些模糊，本章中它的意思是 UCSC 基因组浏览器里的序列。）`biopython`（http://biopython.org/）之类的全能解析工具（比如 `Bio.SeqIO`）由于可以把各种文件格式转化为几种常用内存模型（比如 `Bio.Seq`、`Bio.SeqRecord` 和 `Bio.SeqFeature` 等），深受大家欢迎。

利用 Apache Avro 之类的序列化框架，我们可以把这些问题一并解决掉。Avro 的关键是使数据模型（即显示模式）独立于底层数据存储格式和语言的内存表示：Avro 指定进程之间某种数据的通信方式，不管它是在互联网上跨进程通信，还是进程将数据写入某种文件格式。比如，一个 Java 程序可以使用 Avro 将数据写入多种底层数据格式，但 Avro 的数据模型兼容所有这些格式。这样每个进程都不需要担心不同格式之间的兼容性，而只要知道怎样读取 Avro 数据模型即可，文件系统则知道如何生成 Avro 数据。

我们现在来看一个序列特征的示例。先用 Avro 接口定义语言（IDL）来给对象指定合适的模式：

```
enum Strand {
  Forward,
  Reverse,
  Independent
}

record SequenceFeature {
  string featureId;
  string featureType; ❶
  string chromosome;
  long startCoord;
  long endCoord;
  Strand strand;
  double value;
  map<string> attributes;
}
```

❶ 特征类型，比如“conservation”“centipede”“gene”。

类型 `SequenceFeature` 可用于对保护水平、是否存在发起者或者核糖体结合位点、转录因子结合位点等进行编码。我们可以把它看成 JSON 格式的二进制版本，但 Avro 限制更多，性能也高得多。给定一个数据模式，Avro 规范精确规定对象的二进制编码，这样我们就可以轻易在进程之间（即使进程使用不同编程语言编写）传递对象，可以通过网络通信的方式，也可以通过将对象存储到磁盘的方式。Avro 项目提供处理 Avro 数据的多种语言编码模块，包括 Java、C/C++、Python 和 Perl。除此之外，编程语言还可以按照语言最优的方式将对象存入内存。使数据模型独立于存储格式的做法还提供了另一层灵活性或抽象。为了提高查询速度，我们可以将 Avro 数据序列化为二进制对象（Avro 容器文件）并以列式文件格式存储（比如 Parquet 文件），也可以为了最高的灵活性（牺牲了效率）而将 Avro 数据存为文本形式的 JSON 格式。最后，Avro 支持模式的进化，用户可以按需要随时添加新字段，软件会优雅地处理好新 / 旧版本模式的兼容问题。

总之，Avro 是一个高效的二进制编码。有了它我们就可以轻松修改数据模式，处理多种编程语言产生的数据，并且将数据存成多种数据格式。使用 Avro 模式存储数据后，我们再也不用为越来越多的定制化数据格式操心，同时又能提高计算效率。

序列化 /RPC 框架

开源社区有许多序列化框架。大数据领域用得最多的序列化框架要数 Apache Avro、Apache Thrift 和谷歌公司的 Protocol Buffers（protobuf）。本质上它们都提供了一个 IDL，用于说明对象 / 消息类型的模式，而且都可以编译成许多不同编程语言。Thrift 在 Protocol Buffers 的 IDL 之上还可以指定 RPC（谷歌开源了基于 protobuf 的 RPC 框架 gRPC）。最后在 IDL 和 RPC 之上，Avro 还提供了将数据存储到磁盘上的文件格式规范。要想泛泛地说哪个序列化框架适合哪种场合是不容易的，因为它们都支持不同的语言而且对不同语言的性能也各不相同。谷歌最近发布了一个“序列化”框架，对在线传输（on-the-wire）和内存中（in-memory）使用相同的字节表示，从而有效地消减了昂贵的序列化步骤。

因为不同的框架支持不同的语言，并且对于不同的语言又有不同的性能，所以很难宽泛地说在什么情况下哪个框架最合适。

对实际数据来说，前面示例中的 `SequenceFeature` 模型有些简单，但大数据基因（Big Data Genomics，BDG）项目（http://bdgenomics.org/）已经为我们提供了许多现成对象的 Avro 模式定义，比如：

- 表示读数的 `AlignmentRecord`
- 表示基因组变体和元数据的 `Variant`
- 表示一个基因位点的命名基因型 `Genotype`
- 表示序列特征（基因段标注）的 `Feature`

实际模式可以在 `bdg-formats` 的 GitHub 资料库（https://github.com/bigdatagenomics/bdg-formats）上找到。BDG 格式可以替代无处不在的“传统”格式（如 BAM 和 VCF），但更常用作高性能的“中间”格式。（这些 BDG 格式的最初目标是取代 BAM 和 VCF，但是 BAM 和 VCF 极其广泛的使用已经证明这个目标是很难实现的。）全球基因学和健康联盟也在开始使用 Protocol Buffers 开发自己的模式（https://github.com/ga4gh/schemas）。这应该不会造成 http://xkcd.com/927/ 的状况（这里面有太多相互竞争的模式）。即使出现这种状况，相比目前那些定制的 ASCII 编码，Avro 还是在性能和数据建模方面有巨大优势。本章后面将使用几个 BDG 模式来完成一些典型的基因学任务。

10.2　用ADAM CLI导入基因学数据

本章在 Spark 中大量使用基因学项目 ADAM。该项目还在持续开发之中，包括它的文档也是。如果你碰到问题，一定要检查一下 GitHub 上最新的 README 文件、GitHub 问题跟踪器和 `adam-developer` 邮件列表。

BDG 核心基因学工具称为 ADAM。从一组映射读取（mapped read）开始，这些核心工具提供重复标注（mark-duplicate）、基本质量分数重校（base quality score recalibration，BQSR）、插入和缺失突变重新比对（indel realignment）和变体识别（variant calling）等功能。为了简化这些核心功能的使用，ADAM 还提供了一个命令行界面工具。相比于 HPC，这些命令行工具可以识别 Hadoop 和 HDFS，其中许多工具可以自动在整个集群中进行并行化而不用用户手动拆分文件或调度作业。

按照 README 文件的指示，我们可以构建 ADAM 项目：

```
git clone https://github.com/bigdatagenomics/adam.git && cd adam
export "MAVEN_OPTS=-Xmx512m -XX:MaxPermSize=128m"
mvn clean package -DskipTests
```

或者也可以从 ADAM 的 Github 页面上下载。

ADAM 提供一个作业提交脚本，可以实现与 Spark 的 `spark-submit` 的交互。使用该脚本最简单的方式可能就是给它一个别名：

```
export ADAM_HOME=path/to/adam
alias adam-submit="$ADAM_HOME/bin/adam-submit"
```

现在应该可以从命令行上运行 ADAM 工具并得到如下消息。正如下面的用法介绍所示，Spark 的参数要在 ADAM 的参数之前指定。

```
$ adam-submit
Using ADAM_MAIN=org.bdgenomics.adam.cli.ADAMMain
[...]

       e         888~-_          e             e    e
      d8b        888   \        d8b           d8b  d8b
     /Y88b       888    |      /Y88b         d888bdY88b
    /  Y88b      888    |     /  Y88b       / Y88Y Y888b
   /____Y88b     888   /     /____Y88b     /   YY   Y888b
  /      Y88b    888_-~     /      Y88b   /          Y888b

Usage: adam-submit [<spark-args> --] <adam-args>

Choose one of the following commands:

ADAM ACTIONS
          countKmers : Counts the k-mers/q-mers from a read dataset.
    countContigKmers : Counts the k-mers/q-mers from a read dataset.
           transform : Convert SAM/BAM to ADAM format and optionally perform...
   transformFeatures : Convert a file with sequence features into correspondin...
         mergeShards : Merges the shards of a file
      reads2coverage : Calculate the coverage from a given ADAM file
[...]
```

想要代码成功运行，可能需要设置一些环境变量，如 `SPARK_HOME` 和 `HADOOP_CONF_DIR`。

我们先得到一个 .bam 文件，里面包含一些 mapped NGS read，将它们转换为相应的 BDG 格式（这里也就是 `AlignedRecord`）并保存到 HDFS 上。首先我们取得一个合适的 .bam 文件并把它放到 HDFS 上。

```
# 注意该文件大小有 16GB
curl -O ftp://ftp.ncbi.nih.gov/1000genomes/ftp/phase3/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam

# 也可以用Aspera，它的速度快得多
ascp -i path/to/asperaweb_id_dsa.openssh -QTr -l 10G \
anonftp@ftp.ncbi.nlm.nih.gov:/1000genomes/ftp/phase3/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam .

hadoop fs -put HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam /user/ds/genomics
```

接着可以用 ADAM 转换命令把 .bam 文件转成 Parquet 格式（请参考后面的“Parquet 格式和列式存储”）。该命令既能在集群上运行，也能在本地模式下运行。

```
adam-submit \
  --master yarn \ ❶
  --deploy-mode client \
  --driver-memory 8G \
  --num-executors 6 \
  --executor-cores 4 \
  --executor-memory 12G \
  -- \
  transform \ ❷
  /user/ds/genomics/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam \
  /user/ds/genomics/reads/HG00103
```

❶ 在 YARN 上执行的 Spark 参数示例。

❷ ADAM 命令本身。

这会使控制台产生大量输出，其中包括跟踪作业进度的 URL。我们来看看输出的具体内容：

```
$ hadoop fs -du -h /user/ds/genomics/reads/HG00103
0        0        ch10/reads/HG00103/_SUCCESS
8.6 K    25.7 K   ch10/reads/HG00103/_common_metadata
462.0 K  1.4 M    ch10/reads/HG00103/_metadata
1.5 K    4.4 K    ch10/reads/HG00103/_rgdict.avro
17.7 K   53.2 K   ch10/reads/HG00103/_seqdict.avro
103.1 M  309.3 M  ch10/reads/HG00103/part-r-00000.gz.parquet
102.9 M  308.6 M  ch10/reads/HG00103/part-r-00001.gz.parquet
102.7 M  308.2 M  ch10/reads/HG00103/part-r-00002.gz.parquet
[...]
```

```
106.8 M  320.4 M  ch10/reads/HG00103/part-r-00126.gz.parquet
12.4 M   37.3 M   ch10/reads/HG00103/part-r-00127.gz.parquet
```

结果数据集把 /user/ds/genomics/reads/HG00103/ 目录下所有的文件都合在一起，每个 part-*.parquet 文件对应一个 Spark 任务输出。你可能也会注意到数据的压缩效率比开始的 .bam 文件（底层是 gzip 压缩）要高，这要归功于列式存储（详见后面的“Parquet 格式和列式存储”）：

```
$ hadoop fs -du -h "/user/ds/genomics/HG00103.*.bam"
15.9 G  /user/ds/genomics/HG00103. [...] .bam

$ hadoop fs -du -h -s /user/ds/genomics/reads/HG00103
12.8 G  /user/ds/genomics/reads/HG00103
```

我们在命令行里交互地看一个对象。首先用 ADAM 助手脚本启动 Spark shell。它默认的参数 / 选项与 Spark 脚本相同，但会加载所有必需的 JAR 文件。下面的示例中，Spark 运行在 YARN 上：

```
export SPARK_HOME=/path/to/spark
$ADAM_HOME/bin/adam-shell
[...]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit [...], Java 1.8.0_112)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

注意现在的任务是运行在 YARN 上的，交互式 Spark shell 要求是 `yarn-client` 模式，这时驱动程序在本地运行。同时我们也需要设置好 `HADOOP_CONF_DIR` 或者 `YARN_CONF_DIR`。现在把 aligned read 数据加载为 `RDD[AlignmentRecord]`：

```
import org.bdgenomics.adam.rdd.ADAMContext._

val readsRDD = sc.loadAlignments("/user/ds/genomics/reads/HG00103")
readsRDD.rdd.first()
```

这会输出一些日志和结果本身（为清楚起见，以下输出经过修改）：

```
res3: org.bdgenomics.formats.avro.AlignmentRecord = {
  "readInFragment": 0, "contigName": "1", "start": 9992,
  "oldPosition": null, "end": 10091, "mapq": 25,
  "readName": "SRR062643.12466352",
```

```
  "sequence": "CTCTTCCGATCTCCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAA
CCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCT",
  "qual": "##@@BA:36<FBGCBBD>AHHB@4DD@B;0DEF6A9EDC6>9CCC@9@IIH@I8IIC4
@GH=HGHCIHHHGAGABEGAGG@EGAFHGFFEEE?DEFDDA.",
  "cigar": "1S99M", "oldCigar": null, "basesTrimmedFromStart": 0,
  "basesTrimmedFromEnd": 0, "readPaired": true, "properPair": false,
  "readMapped": true, "mateMapped": false,
  "failedVendorQualityChecks": false, "duplicateRead": false,
  "readNegativeStrand": true, "mateNegativeStrand": true,
  "primaryAlignment": true, "secondaryAlignment": false,
  "supplementaryAlignment": false, "mis...
```

你看到的读数可能不一样，原因是集群上数据的分区不同，不能保证哪条读数会先返回。

现在我们可以在数据集上交互式地提出问题，在问这些问题的同时集群在后台执行运算。数据集中有多少个读数？

```
readsRDD.rdd.count()
...
res16: Long = 160397565
```

接着看看这些数据集中的读数，是来自人类染色体吗？

```
val uniq_chr = (readsRDD.rdd
  .map(_.getContigName)
  .distinct()
  .collect())
uniq_chr.sorted.foreach(println)
...
1
10
11
12
[...]
GL000249.1
MT
NC_007605
X
Y
hs37d5
```

很好！我们观察一下读数，这些读数来自染色体 1~22，*X* 和 *Y*，以及不是“主”染色体的一部分或位置未知的一些其他染色体块。现在来更进一步分析这条语句：

```
val uniq_chr = (readsRDD ❶
  .rdd ❷
  .map(_.getContigName) ❸
  .distinct() ❹
  .collect()) ❺
```

❶ `AlignedReadRDD`：一个 ADAM 类型，包含存储我们所有数据的 RDD。

❷ `RDD[AlignmentRecord]`：底层的 Spark RDD。

❸ `RDD[String]`：从每个 `AlignmentRecord` 对象中提取 contig name 并将其转成字符串。

❹ `RDD[String]`：会产生一个 reduce/shuffle，以将所有不同的 contig name 汇总起来；虽然这个 RDD 应该不大，但它还是一个 RDD。

❺ `Array[String]`：会触发计算并将 RDD 中的数据传到客户端应用（即 shell）。

举一个更具临床意义的例子，假设我们正在测试一个人的基因组，以检查基因中是否携带任何导致儿童患囊性纤维化（cystic fibrosis，CF）风险增加的基因变体。我们的基因测试使用下一代 DNA 测序来生成多个相关基因的读数，如 CFTR 基因（其突变可引起 CF）。在数据流过我们的基因分类管道后，我们确定 CFTR 基因似乎具有破坏其功能的提前终止密码子。然而，这种突变在 HGMD（http://www.hgmd.cf.ac.uk/ac/index.php）中从未出现，也没有在汇聚了 CF 基因变体的 Sickkids CFTR 数据库（http://www.genet.sickkids.on.ca/app）中出现。我们想回过头来看看原始基因序列数据并检查潜在有害基因型是否属于误报。为此需要人工分析变体位点，比如 7 号染色体所在的 117149189 位置对应的所有读数（如图 10-1 所示）：

```
val cftr_reads = (readsRDD.rdd
  .filter(_.getContigName == "7")
  .filter(_.getStart <= 117149189)
  .filter(_.getEnd > 117149189)
  .collect())
cftr_reads.length // cftr_reads 是一个本地的Array[AlignmentRecord]
...
res2: Int = 9
```

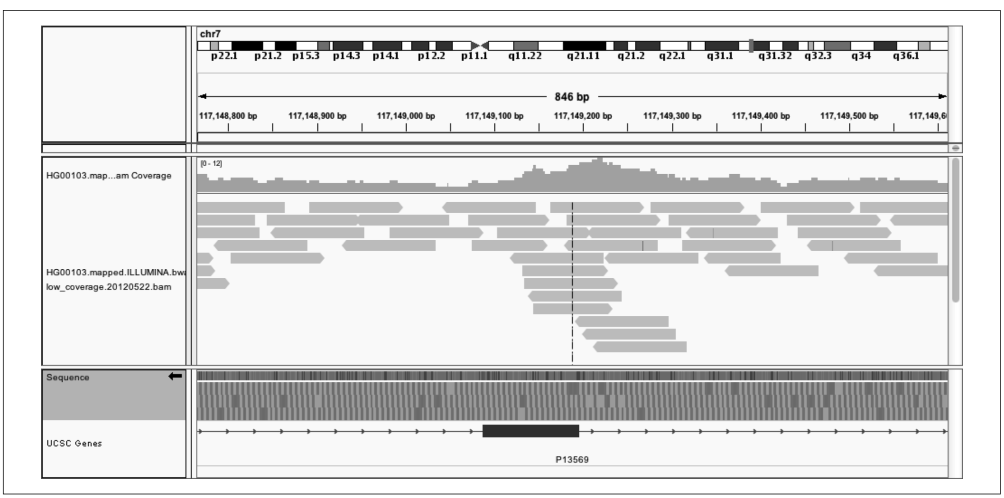


图 10-1：HG00103 在 CFTR 基因中 chr7:117149189 位置的 IGV 图像

现在我们可以人工检查这 9 个读数或者按照指定方式对齐它们，并检查报告的致病变体是否属于误报。下面是个小练习：请问 7 号染色体的平均覆盖率是多少？（这个值肯定很小，它不足以让我们可靠地判断给定未知的基因型。）

假设我们有一个向临床医生提供载体筛选服务的诊断室，用 Hadoop 对原始数据进行归档可以使数据保持在相对较“热”的状态（与磁带等归档技术相比）。除了可靠性高的优点之外，用 Hadoop 处理实际数据还能让我们很便捷地访问所有历史数据，这些历史数据可以用于质量控制（QC）或那些需要人工干预的场合，比如本章前面提到的 CFTR 示例。除了可以快速访问全部数据，数据集中存放后我们还能轻松地进行大规模分析，比如进行人口基因学分析、大规模 QC 分析，等等。

Parquet格式和列式存储

上一节，我们讨论了如何操作大量序列数据而不用担心底层存储规范或运算的并行化。但是，请注意 ADAM 项目用的是 Parquet 文件格式，该格式是这里性能大幅提升的原因。

Parquet 是一种开源文件格式规范，并且它提供了一套 reader/writer 实现。一般情况下对分析型查询用到的数据（一次写入多次读取），我们都推荐使用 Parquet 格式。该格式思想主要来源于谷歌的 Dremel 系统［请参考“Dremel: Interactive Analysis of Web-scale Datasets”（https://static.googleusercontent.com/media/research.google.com/en/pubs/archive/36632.pdf）Proc. VLDB, 2010, by Melnik et al.］中底层的数据存储格式。Parquet 的数据模型可以和 Avro、Thrift 以及 Protocol Buffers 兼容。具体来说，它支持大多数常用数据库类型（整型、双精度、字符串等），也支持数组和记录类型，包括嵌套类型。更重要的是，它是一种列式文件格式，也就是说许多记录的某个列的值在磁盘上是连续存储在一起的（如图 10-2 所示）。这种物理数据布局大幅提升了数据编码 / 压缩的效率，并且通过减少读取 / 反序列化数据的量大幅减少了查询时间（http://the-paper-trail.org/blog/columnar-storage/）。Parquet 中可以为每列指定不同的编码 / 压缩机制，每列都支持 run-length 编码、dictionary 编码和 delta 编码。

Parquet 在提高性能方面另一个有用的功能是**谓词下推**（predicate pushdown）。**谓词**是一个表达式或者函数，它的值可以根据数据记录（也就是 SQL `WHERE` 子句中的表达式）计算出来，要么为 `true`，要么为 `false`。在前面我们的 CFTR 查询中，Spark 必须把每个 `AlignmentRecord` 全部反序列化 / 物化后才能再确定每个 `AlignmentRecord` 是否通过谓词测试。这会导致 I/O 和 CPU 时间的大量浪费。我们可以在 Parquet reader 实现中指定一个谓词类，这样在物化整个记录前我们可以只反序列化那些用于判断的必要列。

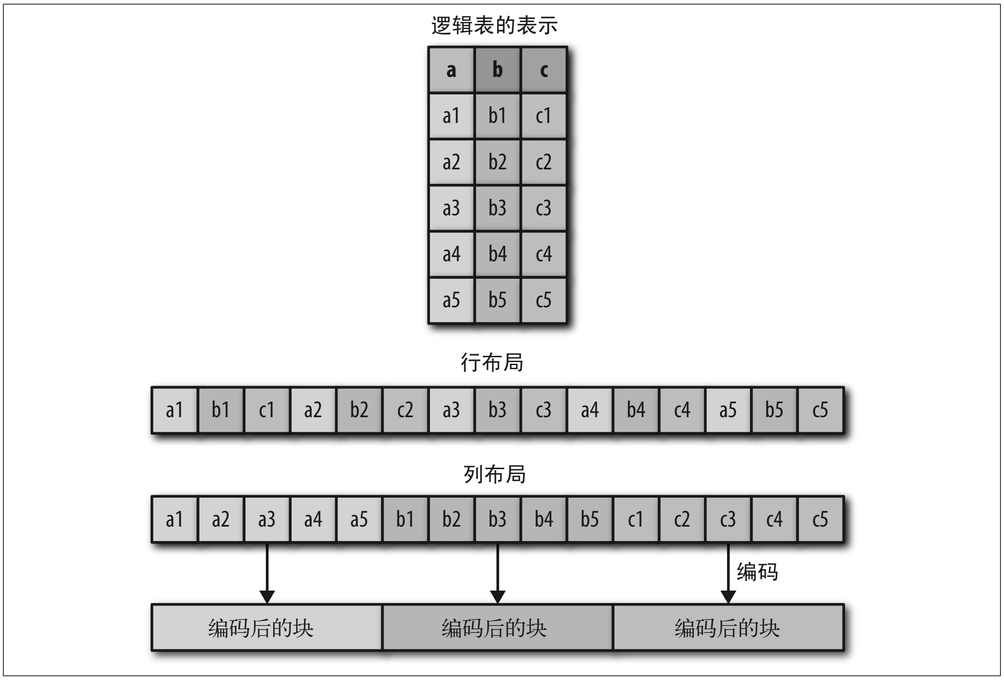


图 10-2：面向行和面向列的布局差异

比如要利用谓词下推技术实现我们的 CFTR 查询，需要先定义一个合适的谓词类，它用于测试 `AlignmentRecord` 是否是目标位点：

```
import org.apache.parquet.filter2.dsl.Dsl._

val chr = BinaryColumn("contigName")
val start = LongColumn("start")
val end = LongColumn("end")

val cftrLocusPredicate = (
  chr === "7" && start <= 117149189 && end >= 117149189) ❶
```

❶ 想了解 DSL 的更多信息，请参阅文档。请注意，我们使用 === 而不是 ==。

因为使用了 Parquet 特定的功能，所以我们必须明确使用 Parquet Reader 加载数据：

```
val readsRDD = sc.loadParquetAlignments(
  "/user/ds/genomics/reads/HG00103",
  Some(cftrLocusPredicate))
```

上述代码执行速度应该更快，因为它不再需要全部物化所有的 `AlignmentRecord` 对象。

10.3 从ENCODE数据预测转录因子结合位点

本例中我们将用公开的序列特征数据来构建一个简单的转录因子结合位点模型。转录因子（TF）是染色体中与特定的 DNA 序列结合的蛋白质，它有助于控制不同基因的表达。因此，转录因子是确定一个细胞的基因型的关键，许多生理学和疾病过程都离不开它。染色质免疫沉淀测序（ChIP-seq）是一种基于 NGS 的实验，可以在基因组范围内描述对某个 TF 在某个细胞 / 组织类型中的位点结合。然而，ChIP-seq 成本高技术难度大，而且需要对每种组织和 TF 的成对组合进行单独实验。相比而言，DNase-seq 实验寻找染色体组内的开放的染色质，它对每种组织类型只做一次。与对每个组织 /TF 组合都进行基于 ChIP-seq 的 TF 结合位点实验不同，我们希望只要能拿到 DNase-seq 数据就可以预测新组织类型中的 TF 结合位点。

更具体地，我们将使用 DNase-seq 数据、已知序列主题数据（来源于 HT-SELEX，http://www.cell.com/cell/fulltext/S0092-8674(12)01496-1）和其他的一些公开的 ENCODE 数据集（https://www.encodeproject.org/）来预测 CTCF 转录因子的结合位点。我们选取了 6 种有 DNase-seq 和 CTCF ChIP-seq 数据的不同细胞类型。训练样本为 DNA 酶超敏（HS）峰值，TF 为绑定 / 未绑定的二进制标签来自 ChIP-seq 数据。

梳理整个数据流：主要的训练 / 测试示例将从 DNase-seq 数据中导出。开放染色质的每个区域（基因组上的区间）将用于预测特定组织类型中的特定 TF 是否被绑定在那里。为此，我们在空间上将 ChIP-seq 数据加入 DNase-seq 数据中；每个重叠都是 DNase-seq 对象的正标签。最后，为了提高预测的准确率，我们在 DNase-seq 数据的每个区间生成了一些额外的特征，比如保存得分（来自 phyloP 数据集），到转录起始位点的距离（使用 Gencode 数据集），以及 DNase-seq 区间的序列与 TF 的已知结合基序（使用凭经验确定的位置权重矩阵）相匹配的程度如何。几乎在任何情况下，通过执行空间连接（可能的聚合）都可以将这些特征添加到训练示例中。

我们将使用如下细胞系数据。

- GM12878
  被广泛研究的淋巴细胞系（lymphoblastoid cell line）。
- K562
  慢性粒细胞白血病细胞系（female chronic myelogenous leukemia）。
- BJ
  皮肤成纤维细胞（skin fibroblast）。
- HEK293
  胚肾细胞系（embryonic kidney）。

- H54

  脑胶质瘤（glioblastoma）。

- HepG2

  肝细胞癌（hepatocellular carcinoma）。

首先我们把 .narrowPeak 格式的细胞系 DNase 数据下载下来：

```
hadoop fs -mkdir /user/ds/genomics/dnase
curl -s -L <...DNase URL...> \ ❶
  | gunzip \ ❷
  | hadoop fs -put - /user/ds/genomics/dnase/sample.DNase.narrowPeak
[...]
```

❶ 实际的 `curl` 命令请参考本书附带的 GitHub 资料库代码。

❷ 流式压缩。

接下来下载 CTCF 转录因子的 ChIP-seq 数据和 GENCODE 数据。ChIP-seq 数据也是 .narrowPeak 格式的，而 GENCODE 数据是 GTF 格式的。

```
hadoop fs -mkdir /user/ds/genomics/chip-seq
curl -s -L <...ChIP-seq URL...> \ ❶
  | gunzip \
  | hadoop fs -put - /user/ds/genomics/chip-seq/samp.CTCF.narrowPeak
[...]
```

❶ 实际的 `curl` 命令请参考本书附带的 GitHub 资料库代码。

注意，在把数据写到 HDFS 上的同时对数据流用 `gunzip` 解压。现在我们下载实际的人类基因组序列，它们被用来评估位置权重矩阵，以生成其中一个特征：

```
# gh19人类基因组序列
curl -s -L -O \
  "http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit"
```

最后，conservation 数据是 fixed wiggle 格式的，不能把它作为一个可拆分文件进行读取。具体而言，在任意位置输入文件并开始读取记录非常困难，因此它不是“可分割的”。这是因为摆动格式在分布数据记录时，使用了描述当前基因组位置的元数据。所以在读取色度坐标元数据时，任务无法知道在文件中要往后读多少数据，我们要在往 HDFS 上写数据的同时使用 BEDOPS 工具，将 .wigFix 转换成 BED 格式。

```
hadoop fs -mkdir /user/ds/genomics/phylop
for i in $(seq 1 22); do
    curl -s -L <...phyloP.chr$i URL...> \ ❶
      | gunzip \
      | wig2bed -d \ ❷
      | hadoop fs -put - "/user/ds/genomics/phylop/chr$i.phyloP.bed"
```

```
done
[...]
```

❶ 实际的 `curl` 命令请参考本书附带的 GitHub 资料库代码。

❷ 这个命令来自 BEDOPS CLI。不过，编写自己的 Python 脚本也很容易。

最后，我们在 Spark shell 中将 phyloP 数据从基于文本的 .bed 格式一次性转换成 Parquet 格式。

```
import org.bdgenomics.adam.rdd.ADAMContext._
(sc
  .loadBed("/user/ds/genomics/phylop_text")
  .saveAsParquet("/user/ds/genomics/phylop"))
```

我们想从所有原始数据生成如下模式的训练数据：

(1) 染色体（chromosome）
(2) 开始位置（start）
(3) 结束位置（end）
(4) 最高 TF 主题 PWM 分数（TF motif PWM score）
(5) 平均 phyloP 保护分数（phyloP conservation score）
(6) 最小 phyloP 保护分数
(7) 最大 phyloP 保护分数
(8) 到最近转录起始位点（TSS）的距离
(9) 转录因子类型（TF identity，本例中一直是 CTCF）
(10) 细胞系（cell line）
(11) 转录因子结合状态（布尔值，目标变量）

这个数据集可以很容易地转换成 `RDD[LabeledPoint]`，然后在机器学习库中使用。我们需要对多个细胞系生成数据，因此对每个细胞系都定义一个 RDD，然后再将它们连接在一起：

```
val cellLines = Vector(
  "GM12878", "K562", "BJ", "HEK293", "H54", "HepG2")
val dataByCellLine = cellLines.map(cellLine => { // 对每个细胞系……
  // ……生成一个RDD以便转换成RDD[LabeledPoint]
})
// 把RDD串在一起之后输入给MLlib等
```

开始之前，我们先加载在整个计算过程中都要用到的一些数据，包括会话转录开始位点、人类基因组参考序列和来自 HT-SELEX（http://www.cell.com/cell/fulltext/S0092-8674(12)01496-1）的 CTCF PWM。我们还定义了一些用于生成 PWM 和 TSS 功能的实用函数：

```
val hdfsPrefix = "/user/ds/genomics"
val localPrefix = "/user/ds/genomics"
```

```
// 为计算特征设置一些广播变量，以及定义一些功能函数

// 加载人类基因组参考序列
val bHg19Data = sc.broadcast(
  new TwoBitFile(
    new LocalFileByteAccess(
      new File(Paths.get(localPrefix, "hg19.2bit").toString))))

// 函数用于寻找最近的转录起始位点
// 傻瓜办法；读者的练习题：让这个函数变得更快
def distanceToClosest(loci: Vector[Long], query: Long): Long = {
  loci.map(x => math.abs(x - query)).min
}

// https://dx.doi.org/10.1016/j.cell.2012.12.009的CTCF PWM
// 使用genomics/src/main/python/pwm.py生成
val bPwmData = sc.broadcast(Vector(
  Map('A'->0.4553,'C'->0.0459,'G'->0.1455,'T'->0.3533),
  Map('A'->0.1737,'C'->0.0248,'G'->0.7592,'T'->0.0423),
  Map('A'->0.0001,'C'->0.9407,'G'->0.0001,'T'->0.0591),
  Map('A'->0.0051,'C'->0.0001,'G'->0.9879,'T'->0.0069),
  Map('A'->0.0624,'C'->0.9322,'G'->0.0009,'T'->0.0046),
  Map('A'->0.0046,'C'->0.9952,'G'->0.0001,'T'->0.0001),
  Map('A'->0.5075,'C'->0.4533,'G'->0.0181,'T'->0.0211),
  Map('A'->0.0079,'C'->0.6407,'G'->0.0001,'T'->0.3513),
  Map('A'->0.0001,'C'->0.9995,'G'->0.0002,'T'->0.0001),
  Map('A'->0.0027,'C'->0.0035,'G'->0.0017,'T'->0.9921),
  Map('A'->0.7635,'C'->0.0210,'G'->0.1175,'T'->0.0980),
  Map('A'->0.0074,'C'->0.1314,'G'->0.7990,'T'->0.0622),
  Map('A'->0.0138,'C'->0.3879,'G'->0.0001,'T'->0.5981),
  Map('A'->0.0003,'C'->0.0001,'G'->0.9853,'T'->0.0142),
  Map('A'->0.0399,'C'->0.0113,'G'->0.7312,'T'->0.2177),
  Map('A'->0.1520,'C'->0.2820,'G'->0.0082,'T'->0.5578),
  Map('A'->0.3644,'C'->0.3105,'G'->0.2125,'T'->0.1127)))

// 根据TF PWM计算一个主题分数
def scorePWM(ref: String): Double = {
  val score1 = (ref.sliding(bPwmData.value.length)
    .map(s => {
      s.zipWithIndex.map(p => bPwmData.value(p._2)(p._1)).product})
    .max)
  val rc = Alphabet.dna.reverseComplementExact(ref)
  val score2 = (rc.sliding(bPwmData.value.length)
    .map(s => {
      s.zipWithIndex.map(p => bPwmData.value(p._2)(p._1)).product})
    .max)
  math.max(score1, score2)
}

// 构建一个in-memory的数据结构来计算到TSS的距离
// 本质上我们在这里手工实现了一个广播连接
val tssRDD = (
  sc.loadFeatures(
    Paths.get(hdfsPrefix, "gencode.v18.annotation.gtf").toString).rdd
```

```
    .filter(_.getFeatureType == "transcript")
    .map(f => (f.getContigName, f.getStart))).rdd
    .filter(_.getFeatureType == "transcript")
    .map(f => (f.getContigName, f.getStart)))
  // 这个广播变量将只有“连接”中被广播的那一边
  val bTssData = sc.broadcast(tssRDD
    // 按contigName分组
    .groupBy(_._1)
    // 为每个染色体创建TSS位点的Vector
    .map(p => (p._1, p._2.map(_._2.toLong).toVector))
    // 使用collect方法将数据取回到本地内存中，再广播出去
    .collect().toMap)

  // 加载保存的数据；独立于细胞系
  val phylopRDD = (
    sc.loadParquetFeatures(Paths.get(hdfsPrefix, "phylop").toString).rdd
    // 清理phyloP数据中的一些不规则记录
    .filter(f => f.getStart <= f.getEnd)
    .map(f => (ReferenceRegion.unstranded(f), f))).rdd
    // 清理phyloP数据中的一些不规则记录
    .filter(f => f.getStart <= f.getEnd)
    .map(f => (ReferenceRegion.unstranded(f), f)))
```

现在已经加载了训练示例所需的数据，我们定义一个在每个细胞系上进行数据计算的“loop”循环体。注意，读取的是文本形式的 ChIP-seq 和 DNase 数据，因为数据集不是特别大，所以对性能影响不大。

首先我们把 DNase 和 ChIP-seq 数据加载为 RDD：

```
  val dnasePath = (
    Paths.get(hdfsPrefix, s"dnase/$cellLine.DNase.narrowPeak")
      .toString)
  val dnaseRDD = (sc.loadFeatures(dnasePath).rdd
    .map(f => ReferenceRegion.unstranded(f))
    .map(r => (r, r))) ❶

  val chipseqPath = (
    Paths.get(hdfsPrefix, s"chip-seq/$cellLine.ChIP-seq.CTCF.narrowPeak")
      .toString)
  val chipseqRDD = (sc.loadFeatures(chipseqPath).rdd
    .map(f => ReferenceRegion.unstranded(f))
    .map(r => (r, r))) ❶
```

❶ `RDD[(ReferenceRegion, ReferenceRegion)]`

核心对象是一个 DNase 高敏位点，它由 `dnaseRDD` 中的 `ReferenceRegion` 对象所定义。交叉 ChIP-seq 峰值位点是由 `chipseqRDD` 中的 `ReferenceRegion` 所定义，它具有 TF 结合位点。因此它被标记为真，其余位点被标记为假。这是使用 ADAM API 提供的一维空间连接原语完成的，连接功能需要一个以 `ReferenceRegion` 作为 key 的 RDD，并根据通常的连接语义（例如，是内部连接还是外部连接）产生具有交叉区域的元组。

```
val dnaseWithLabelRDD = (
  LeftOuterShuffleRegionJoin(bHg19Data.value.sequences, 1000000, sc)
    .partitionAndJoin(dnaseRDD, chipseqRDD) ❶
    .map(p => (p._1, p._2.size)) ❷
    .reduceByKey(_ + _) ❸
    .map(p => (p._1, p._2 > 0)) ❹
    .map(p => (p._1, p))) ❺
```

❶ RDD[(ReferenceRegion, Option[ReferenceRegion])]：这里有一个 Option，因为我们正在使用左外连接。

❷ RDD[(ReferenceRegion, Int)]：0 表示 None，1 表示成功匹配。

❸ 聚集所有可能交叉 DNase 位点的 TF 结合位点。

❹ 正值表示数据集之间有重叠，因此是 TF 结合位点。

❺ 把 ReferenceRegion 变成 key 来为下一个连接准备 RDD。

另外，我们通过连接 DNase 数据和 phyloP 数据来计算保存特征：

```
// 给定位点上的phyloP值并计算
def aggPhylop(values: Vector[Double]) = {
  val avg = values.sum / values.length
  val m = values.min
  val M = values.max
  (avg, m, M)
}
val dnaseWithPhylopRDD = (
  LeftOuterShuffleRegionJoin(bHg19Data.value.sequences, 1000000, sc)
    .partitionAndJoin(dnaseRDD, phylopRDD) ❶
    .filter(!_._2.isEmpty) ❷
    .map(p => (p._1, p._2.get.getScore.doubleValue))
    .groupByKey() ❸
    .map(p => (p._1, aggPhylop(p._2.toVector)))) ❹
```

❶ RDD[(ReferenceRegion, Option[Feature])]。

❷ 过滤缺少 phyloP 数据的位点。

❸ 将每个位点的所有 phyloP 值汇总在一起。

❹ RDD[(ReferenceRegion, (Double, Double, Double))]。

现在我们计算每个 DNase 峰值的最终特征集合，通过将上面的两个 RDD 连接在一起，并通过与位点的映射添加一些附加特征：

```
// 构建最终训练样本RDD
val examplesRDD = (
  InnerShuffleRegionJoin(bHg19Data.value.sequences, 1000000, sc) ❶
    .partitionAndJoin(dnaseWithLabelRDD, dnaseWithPhylopRDD)
    .map(tup => {
      val seq = bHg19Data.value.extract(tup._1._1) ❷
      (tup._1, tup._2, seq)})
```

```
.filter(!_._3.contains("N")) ❸
.map(tup => { ❹
  val region = tup._1._1
  val label = tup._1._2
  val contig = region.referenceName
  val start = region.start
  val end = region.end
  val phylopAvg = tup._2._1
  val phylopMin = tup._2._2
  val phylopMax = tup._2._3
  val seq = tup._3
  val pwmScore = scorePWM(seq)
  val closestTss = math.min(
    distanceToClosest(bTssData.value(contig), start),
    distanceToClosest(bTssData.value(contig), end))
  val tf = "CTCF"
  (contig, start, end, pwmScore, phylopAvg, phylopMin, phylopMax,
    closestTss, tf, cellLine, label)}))
```

❶ 内部连接确保我们得到定义完善的特征向量。

❷ 提取基因组中与该位点对应的基因组序列，并附加到元组中。

❸ 丢弃任何基因组序列模糊的位点。

❹ 这里是我们最终建立的特征向量。

这个最终的 RDD 在遍历细胞系的每次循环中都要计算一次。最后我们把每个细胞系的 RDD 结合在一起，并且缓存在内存中为模型训练做准备：

```
val preTrainingData = dataByCellLine.reduce(_ ++ _)
preTrainingData.cache()

preTrainingData.count() // 802059
preTrainingData.filter(_._12 == true).count() // 220344
```

现在为了训练分类器，我们可以对 `preTrainingData` 中的数据进行归一化并将其转换成 `RDD[LabeledPoint]`，详细情况可以参考第 4 章。注意，这里要执行交叉验证，应该在每个 fold 中取出一个细胞系用于验证。

10.4 查询1000 Genomes项目中的基因型

在这个示例中，我们要导入全部 1000 Genomes 基因型数据集。我们先把原始数据下载下来并直接存放到 HDFS 上，解压，然后运行 ADAM 作业将数据转成 Parquet 格式。下面的示例命令应该针对所有染色体运行，它在整个集群中并行执行：

```
curl -s -L ftp://.../1000genomes/.../chr1.vcf.gz \ ❶
  | gunzip \
  | hadoop fs -put - /user/ds/genomics/1kg/vcf/chr1.vcf ❷

adam/bin/adam-submit --master yarn --deploy-mode client \
```

```
    --driver-memory 8G --num-executors 192 --executor-cores 4 \
    --executor-memory 16G \
    -- \
    vcf2adam /user/ds/genomics/1kg/vcf /user/ds/genomics/1kg/parquet
```

❶ 实际的 curl 命令请参考本书附带的 GitHub 资料库代码。

❷ 将文本 VCF 文件复制到 Hadoop 上。

接着在 ADAM shell 中加载并检查对象，代码如下：

```
import org.bdgenomics.adam.rdd.ADAMContext._

val genotypesRDD = sc.loadGenotypes("/user/ds/genomics/1kg/parquet")
val gt = genotypesRDD.rdd.first()
...
```

例如对每个与 CTCF 绑定点重叠的基因变体，计算在所有样本上少数等位基因出现的频率。本质上，我们需要把上一节中的 CTCF 数据和 1000 Genomes 项目中的基因型数据进行联结。在之前 TF 结合位点的例子中，我们展示了如何直接使用 ADAM 连接机制。然而在很多情况下，当通过 ADAM 加载数据时，我们获得了一个实现 GenomicRDD 特征的对象，它具有一些内置的过滤和连接方法，如下所示：

```
import org.bdgenomics.adam.models.ReferenceRegion
import org.bdgenomics.adam.rdd.InnerTreeRegionJoin
val ctcfRDD = (sc.loadFeatures(
  "/user/ds/genomics/chip-seq/GM12878.ChIP-seq.CTCF.narrowPeak").rdd
  .map(f => { ❶
    f.setContigName(f.getContigName.stripPrefix("chr"))
    f
  })
  .map(f => (ReferenceRegion.unstranded(f), f)))
val keyedGenotypesRDD = genotypesRDD.rdd.map(f => (ReferenceRegion(f), f))
val filteredGenotypesRDD = ( ❷
  InnerTreeRegionJoin().partitionAndJoin(ctcfRDD, keyedGenotypesRDD)
  .map(_._2))
filteredGenotypesRDD.cache() ❸
filteredGenotypesRDD.count() // 408107700
```

❶ 我们必须执行这个映射，因为 CTCF 数据使用“chr1”，而基因型数据使用“1”来指代相同的染色体。

❷ 使用内连接进行过滤。我们广播 CTCF 数据，因为它比较小。

❸ 由于计算量大，我们会缓存过滤后的数据，以避免重新计算。

我们还需要一个输入为 Genotype 并计算参考 / 替换等位基因个数的函数：

```
import scala.collection.JavaConverters._
import org.bdgenomics.formats.avro.{Genotype, Variant, GenotypeAllele}
def genotypeToAlleleCounts(gt: Genotype): (Variant, (Int, Int)) = {
  val counts = gt.getAlleles.asScala.map(allele => { allele match {
```

```
      case GenotypeAllele.REF => (1, 0)
      case GenotypeAllele.ALT => (0, 1)
      case _ => (0, 0)
    }}).reduce((x, y) => (x._1 + y._1, x._2 + y._2))
    (gt.getVariant, (counts._1, counts._2))
  }
```

最后生成一个 `RDD[(Variant, (Int, Int))]` 并进行汇总：

```
val counts = filteredGenotypesRDD.map(gt => { ❶
  val counts = gt.getAlleles.asScala.map(allele => { allele match {
    case GenotypeAllele.REF => (1, 0)
    case GenotypeAllele.ALT => (0, 1)
    case _ => (0, 0)
  }}).reduce((x, y) => (x._1 + y._1, x._2 + y._2))
  (gt.getVariant, (counts._1, counts._2))
})
val countsByVariant = counts.reduceByKey(
  (x, y) => (x._1 + y._1, x._2 + y._2))
val mafByVariant = countsByVariant.map(tup => {
  val (v, (r, a)) = tup
  val n = r + a
  (v, math.min(r, a).toDouble / n)
})
```

❶ 我们编写的这个函数是匿名的，因为在使用 shell 的过程中可能会遇到闭包序列化的问题。 Spark 将尝试序列化 shell 中的所有内容，而这可能会出错。作为提交的任务运行时，被命名的函数应该工作正常。

RDD `countsByVariant` 存储类型为 `(Variant, (Int, Int))` 的对象，其中元组的第一个成员是特定的基因组变体，第二个成员是一对计数：第一个计数是参考等位基因的数量，而第二个是所看到的候补等位基因的数目。RDD `mafByVariant` 存储类型为 `(Variant, Double)` 的对象，其中第二个成员来自计算 `countsByVariant` 的键 – 值对得到的次要等位基因频率。举个例子：

```
scala> countsByVariant.first._2
res21: (Int, Int) = (1655,4)

scala> val mafExample = mafByVariant.first
mafExample: (org.bdgenomics.formats.avro.Variant, Double) = [...]

scala> mafExample._1.getContigName -> mafExample._1.getStart
res17: (String, Long) = (X,149849811)

scala> mafExample._2
res18: Double = 0.0024110910186859553
```

遍历整个数据集是个大型操作。因为我们只用到基因型数据中的几个字段，所以进行谓词下推和投影肯定是有帮助的，这可以作为练习留给读者自行完成。如果没有合适的群集，请尝试在数据文件的子集上运行计算。

10.5 小结

许多基因学方面的计算都很适合用 Spark 计算模式处理。如果进行实地分析，那么 ADAM 这样的项目最有价值的贡献就是提供了一组表示底层分析对象（及其转换工具）的 Avro 模式。本章中我们看到，只要将数据转换成相应的 Avro 模式，许多大规模计算就比较容易表达和并行化。

虽然基于 Hadoop/Spark 进行科学研究的工具可能相对较少，但现在已经有一些现成的项目可用，我们不必再重新发明轮子。本章我们研究了 ADAM 提供的核心功能，但这个项目已经实现了整个 GATK 最佳实践中的管道任务，包括 BQSR、插入和缺失突变重新比对、去重。除了 ADAM 之外，许多机构都已加入全球基因学和健康联盟，这个组织也开始提供它自己的基因分析模式。Broad Institute 目前正在使用 Spark 开发主要的软件项目，包括最新版本的 GATK4（https://github.com/broadinstitute/gatk）和一个名为 Hail 的新项目（https://github.com/hail-is/hail），用于大规模人口遗传学的计算。西奈山医学院的振荡实验室开发了一组主要用于致癌基因变体研究的 Guacamole 工具（https://github.com/hammerlab/guacamole）。所有这些工具都是开源的，大家可以自由使用。如果你在工作中用到了这些工具，别忘了把改进建议也回馈给社区哦！

第 11 章

基于PySpark和Thunder的神经图像数据分析

作者：于里·莱瑟森

人类大脑像一坨凉粥，但我们对此并不感兴趣。

——阿兰·图灵

随着影像设备和自动化领域的技术发展，大脑功能数据也急剧增长。过去的实验只能靠在头上放几个电极来收集大脑产生的时间序列数据，或者只能拿到大脑的几张静态截面图像，而今天的技术能够在一个不小的机体活跃区域里，在大量神经元上采集大脑活动数据。BRAIN 计划（https://www.braininitiative.nih.gov/）是受美国政府资助的科研计划，旨在推动技术进步，并且制定了宏伟的目标，其中一个目标是在很长一段时间内同时记录老鼠大脑内每个神经元的电子活动。测量技术方面的突破固然重要，但我们认为该计划产生的数据量将开创生物学研究的新模式。

本章将介绍 PySpark API（https://spark.apache.org/docs/latest/api/python/）。有了该 API，我们可以通过 Python 与 Spark 交互。本章还会介绍 Thunder 项目（http://thunder-project.org/），它构建在 PySpark 之上，目的是处理海量时间序列数据，特别是处理神经影像数据。PySpark 是一个特别灵活的工具，可以帮我们进行探索式的大数据分析，它紧密集成 PyData 生态系统的其他工具，包括可视化工具 matplotlib，甚至是“可执行文档”工具 IPython Notebook（Jupyter）。

利用这些工具可以在一定程度上了解斑马鱼的大脑结构。利用 Thunder 可以对斑马鱼大脑的不同区域（代表不同神经元群组）进行聚类，这样就可以找到斑马鱼随时间变化的大脑活动模式。Thunder 是建立在 PySpark RDD API 上的，我们将继续使用它。

11.1 PySpark简介

Python 具有高级语法并且有很多工具包可用，所以很多数据科学家都喜欢用 Python。虽然传统上 Python 语言很难和 JVM 集成，但鉴于 Python 对于数据分析的重要性，Spark 生态系统开始致力于开发 Spark 的 Python API。

Python 与科学计算和数据科学

在科学计算和数据科学领域，人们更喜欢 Python 工具。许多基于 MATLAB、R 或 Mathematica 的传统应用都迁移到 Python 之上了。究其原因，我们总结出如下几个方面：

- Python 是一门高级语言，使用简单，学起来也容易；
- Python 包含了大量的工具包，从小众的数值计算到网页抓取工具再到数据可视化工具，它无所不包；
- Python 可以便捷地和 C/C++ 进行交互，这样人们就可以使用 C/C++ 的高性能工具包，比如 BLAS/LAPACK/ATLAS 等。

这里有几个工具尤其需要读者记住。

- `numpy/scipy/matplotlib`

  这三个工具提供了 MATLAB 的典型功能，包括快速矩阵运算、科学计算函数，还提供了绘图工具，这些绘图工具被广泛使用，其思想也源于 MATLAB。

- `pandas`

  该工具的功能和 R 的 `data.frame` 类似，但启动效率往往要比 `data.frame` 高不少。

- `scikit-learn/statsmodels`

  这两个工具提供了高质量的机器学习算法（分类 / 回归、聚类、矩阵分解等）的实现和统计模型实现。

- `nltk`

  一个深受欢迎的自然语言工具。

可以在 GitHub 上的 awesome-python 代码库（https://github.com/vinta/awesome-python）上找到大量其他 Python 工具包。

启动 PySpark 与启动 Spark 一样：

```
export PYSPARK_DRIVER_PYTHON=ipython # PySpark 也可使用IPython shell
export PYSPARK_PYTHON=path/to/desired/python # 在工作节点上
pyspark --master ... --num-executors ...  ❶
```

❶ pyspark 的输入参数和 Spark 的 spark-submit 以及 spark-shell 参数一样。

可以用 spark-submit 来提交 Python 脚本，spark-submit 能根据脚本文件的扩展名 .py 来识别脚本。你可以指定 driver（如 IPython）和工作节点的 Python 版本；driver 和工作节点的版本必须匹配。当 Python 脚本启动时，它会创建一个 Python 的 SparkContext 对象（命名为 sc），我们通过该对象和集群交互。创建好 SparkContext 之后，PySpark API 的用法和 Scala API 非常类似。比如，要加载 CSV 数据，我们可以这样做：

```
raw_data = sc.textFile('path/to/csv/data') # RDD[string]
# 对数据进行过滤，按逗号进行拆分并解析浮点数以得到RDD[list[float]]
data = (raw_data
    .filter(lambda x: x.startswith("#"))
    .map(lambda x: map(float, x.split(','))))
data.take(5)
```

和 Scala RDD API 一样，我们先加载一个文本文件，过滤掉其中以 # 开头的行，然后将 CSV 数据解析为一个浮点数列表。如在上面的示例代码中，我们向 filter 和 map 传递函数参数所示，Python 中把函数作为参数进行传递的方式是非常灵活的。传递函数时需要将一个 Python 对象作为输入并且返回一个 Python 对象（对于示例代码中的 filter 函数而言，返回值为布尔值）。传递函数时唯一的限制是 Python 对象必须能用 cloudpickle 序列化（cloudpickle 包含了匿名 lambda 函数），并且函数闭包引用的任何模块都必须能在 Python 执行器进程的 PYTHONPATH 中找到。为了确保 Python 执行器进程找到这些模块，要么在整个集群上安装这些模块并在 Python 执行器进程的 PYTHONPATH 中加入它们，要么由 Spark 显式分发相应的 ZIP/EGG 模块文件，这样 Spark 会在分发之后将它们加入 PYTHONPATH 中。显式分发可以通过调用 sc.addPyFile() 来实现。

PySpark RDD 只是 Python 对象的 RDD。和 Python 列表一样，PySpark RDD 可以存储混合类型对象（因为底层的所有对象都是 PyObject 类型的）。

深入PySpark

为了简化调试，同时也为了让读者了解可能的性能陷阱，有必要介绍一些 PySpark 的底层实现，请参见图 11-1。

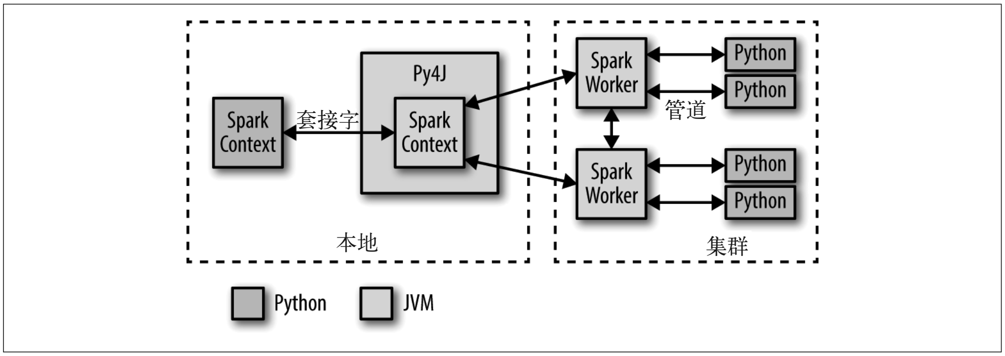


图 11-1：PySpark 内部架构

PySpark 的 Python 解释器在启动时会同时启动一个 JVM，Python 解释器与 JVM 进程之间通过套接字保持通信。PySpark 利用 Py4J 项目来处理 Python 解释器和 JVM 之间的通信。JVM 作为实际的 Spark 驱动程序会加载一个 `JavaSparkContext`，`JavaSparkContext` 和集群中的 Spark 执行器通信。接着，对 `SparkContext` 对象的 Python API 调用会被翻译为对 `JavaSparkContext` 对象的 Java API 调用。举个例子，PySpark 的 `sc.textFile()` 实现将调用分派给 `JavaSparkContext` 的 `.textFile` 方法，该方法最终与 Spark 执行器的 JVM 通信，从而实现从 HDFS 上加载文本数据。

集群上的 Spark 执行器为每个 CPU 核启动一个 Python 解释器，并在需要执行用户代码时通过 Unix 管道 `stin` 以及 `stdout` 与这个解释器进行数据通信。在本地 PySpark 客户端的 Python RDD 对应于本地 JVM 内的一个 `PythonRDD` 对象。和 RDD 相关的数据实际上是以 Java 对象的形式保存在 Spark 的 JVM 中的。举例来说，在 Python 解释器中运行 `sc.textFile()` 将会调用 `JavaSparkContext` 的 `textFile` 方法，该方法会把集群中的数据加载为 Java `String` 对象。类似地，用 `newAPIHadoopFile` 加载一个 Parquet/Avro 文件会把对象加载为 Java Avro 对象。

如果是在 Python RDD 上调用 API，所有相关代码（即 Python 的 lambda 函数）将通过 `cloudpickle` 进行序列化并分发到执行器上。接着，代码的序列化数据由 Java 对象转成 Python 兼容的表示形式（即 pickle 对象）并通过一个管道以流的方式传给执行器相关的 Python 解释器。所有需要 Python 处理的内容都在解释器中执行，结果数据又反过来存储为 JVM 中的 RDD（默认为 pickle 对象）。

相比 Scala，Python 对可执行代码序列化的内置支持不算强大。因此 PySpark 的作者使用一个定制化模块 cloudpickle。cloudpickle 由 PiCloud 项目开发，不过 PiCloud 现在已不活跃了。

11.2 Thunder工具包概况和安装

Thunder 示例和文档

Thunder 包的文档和教程写得非常好。下面的示例引自 Thunder 教程和文档所提供的数据集。

Thunder 是 Spark 上的一个的 Python 工具集，用于处理大型空间 / 时间数据集（即大型多维矩阵）。Thunder 大量使用 NumPy 进行矩阵运算，同时也大量使用 MLlib 工具来实现某些分布式统计技术。由于基于 Python，所以 Thunder 非常灵活而且用户很广。在接下来的一节，我们将介绍 Thunder API 并利用 MLlib 的 K 均值算法实现对神经轨迹进行聚类，这里的 K 均值算法实现是经过 Thunder 和 PySpark 包装过的版本。安装 Thunder 非常简单，运行 `pip install thunder-python` 命令即可，尽管必须在所有工作节点上安装它。

安装并设置完 `SPARK_HOME` 环境之后，就可以创建 PySpark shell 了：

```
$ export PYSPARK_DRIVER_PYTHON=ipython # 像往常一样推荐
$ pyspark --master ... --num-executors ...

[...some logging output...]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Python version 2.7.6 (default, Apr  9 2014 11:54:50)
SparkContext available as sc.

Running thunder version 0.5.0_dev
A thunder context is available as tsc

In [1]:
```

11.3 用Thunder加载数据

Thunder 在设计的时候特别考虑了神经影像数据集，因此比较适合分析那些通常随时间变化的大型影像数据集。

我们先加载样例数据集中的一些斑马鱼大脑图像。这些样例数据来自 Thunder 资料库，目录为 s3://thunder-sample-data/images/fish/。为了演示方便，本章示例的数据集只是原数据集的很小一部分。要了解其他（或更大的）数据集的更多信息，请参阅 Thunder 文档。斑马鱼是生物学研究普遍采用的模式生物，它个体小，繁殖快，可用作脊椎动物培育的模式生

物。人们对斑马鱼的兴趣也源自它超快的繁殖能力。由于斑马鱼是透明的，大脑不大，在神经科学研究中基本上可以对其整个大脑摄取图像，而且这些图像的分辨率高到足以区分个体神经元的程度。下面是加载数据集的代码：

```
import thunder as td
data = td.images.fromtif('/user/ds/neuro/fish', engine=sc) ❶
print data
print type(data.values)
print data.values._rdd
...
Images
mode: spark ❷
dtype: uint8
shape: (20, 2, 76, 87)
<class 'bolt.spark.array.BoltArraySpark'> ❸
PythonRDD[2] at RDD at PythonRDD.scala:48 ❹
```

❶ 请注意传递 `SparkContext` 对象的方式。Thunder 也支持使用同样的 API 操作本地文件。

❷ 我们可以看到一个由 Spark 支持的 `Images` 对象。

❸ 底层的数据容器抽象是一个 `BoltArray`。该项目为本地数据的表示和 Spark RDD 的表示提供了抽象。

❹ 底层的 RDD 对象。

这创建了一个 `Images` 对象，它最终封装了一个 RDD，它可以通过 `data.values.rdd` 来访问。`Images` 对象对外也提供了几个相似的相关功能（比如 `count`、`first` 等）。在 `Images` 内部，对象以键 – 值对的形式存放。

```
print data.values._rdd.first()
...
((0,), array([[[26, 26, 26, ..., 26, 26, 26],
        [26, 26, 26, ..., 26, 26, 26],
        [26, 26, 26, ..., 27, 27, 26],
        ...,
        [26, 26, 26, ..., 27, 27, 26],
        [26, 26, 26, ..., 27, 26, 26],
        [25, 25, 25, ..., 26, 26, 26]],

       [[25, 25, 25, ..., 26, 26, 26],
        [25, 25, 25, ..., 26, 26, 26],
        [26, 26, 26, ..., 26, 26, 26],
        ...,
        [26, 26, 26, ..., 26, 26, 26],
        [26, 26, 26, ..., 26, 26, 26],
        [25, 25, 25, ..., 26, 26, 26]]], dtype=uint8))
```

键 `(0,)` 对应数据集中第零个图像（按数据目录的字母顺序排列），值是对应图像的一个 NumPy 数组。Thunder 中所有的核心数据类型最终都是键 – 值对的 Python RDD，其中键是某种元组，而值为 NumPy 数组。即使 PySpark 通常允许异构集合的 RDD，RDD 中所有

键和值的类型也都相同。由于这种同构性，`Images` 对象对外提供了描述底层维度的一个属性 `.shape`：

```
print data.shape
...
(20, 2, 76, 87)
```

这段代码描述的是 20 个“图像”，其中每个图像都是一个 2 × 76 × 87 的叠层。

像素、体元和叠层

“像素”（pixel）一词是“图像元素”（picture element）两个词构成的合成词。数字图像可以简单建模为二维矩阵，矩阵中每个元素即为一个像素，其值代表颜色的强度（一张彩色图片需要三个这样的矩阵来表示，分别代表红色、绿色和蓝色）。但大脑是三维的，单个二维切面很难捕捉大脑的活动。有多种技术可以处理这个问题，有的将不同平面上的多个二维图像堆叠在一起（即一个 *z* 叠层），有的甚至直接生成三维信息（比如光场显微技术）。这最终会产生一个三维的强度矩阵，每个值代表一个立体元素或体元。同样，Thunder 根据特定的数据类型也把所有图像建模成二维或三维矩阵，并且能够识别像 .tiff 这样的文件格式，.tiff 格式能原生地表示三维叠层。

用 Python 写代码的一个特点就是我们在操作 RDD 时能轻松进行可视化。这里我们使用功能强大的 matplotlib 工具包（见图 11-2）。

```
import matplotlib.pyplot as plt
img = data.first()
plt.imshow(img[:, :, 0], interpolation='nearest', aspect='equal',
    cmap='gray')
```

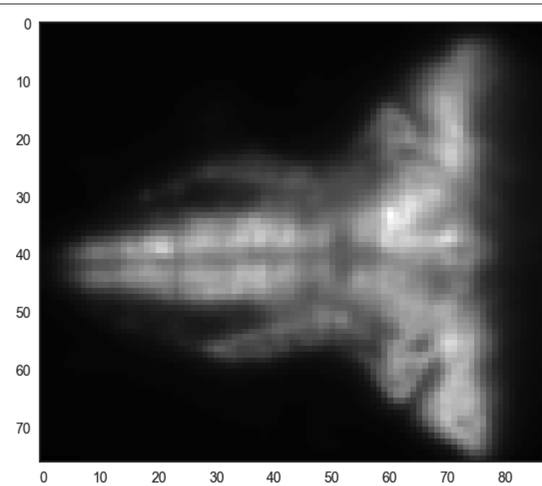


图 11-2：原生斑马鱼数据的一个切面

Images API 提供了强大的分布式图像处理能力，比如对每个图像进行二次采样（如图 11-3 所示）：

```
subsampled = data.subsample((1, 5, 5)) ❶
plt.imshow(subsampled.first()[:, :, 0], interpolation='nearest',
    aspect='equal', cmap='gray')
print subsampled.shape
...
(20, 2, 16, 18)
```

❶ 第一行我们直接对 3 个维度进行采样，这里只用了一行代码：第一维没有做二次采样，而第二维和第三维是每 5 个值取一个。注意这是一个 RDD 操作，所以该行代码立即返回，只有等到出现 RDD 行动时才触发实际的计算。

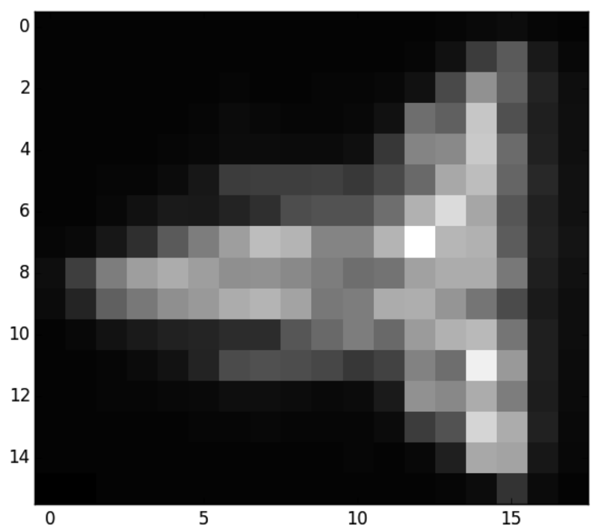


图 11-3：对斑马鱼数据进行二次采样得到的一个切面

虽然分析图像集合对某些操作是有用的（比如对图像进行某种归一化），但它很难处理图像之间的时间关系。

```
series = data.toseries()
```

这个操作把数据大规模重组为一个 Series 对象。Series 是一个键 – 值对的 RDD，键是每个图像的坐标元组（也就是体元标识符），值是一个代表时间序列的一维 NumPy 数组：

```
print series.shape
print series.index
print series.count()
...
```

```
(2, 76, 87, 20)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
13224
```

data 包含 20 个带有维度（76×87×2）的图像，series 包含 13 224（76×87×2）个长度为 20 的时间序列。series.index 属性是一个 Pandas 风格的索引，可以用它引用每个数组。因为原始图像是三维的，所以键是三元组：

```
print series.values._rdd.takeSample(False, 1)[0]
...
((0, 50, 6), array([29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
       29, 29, 29, 29, 29, 29, 29], dtype=uint8))
```

Series API 提供了许多时序运算方法，这些方法可以在单个时间序列上进行计算，也可以对所有的时间序列进行计算，比如：

```
print series.max().values
...
[[[[158 152 145 143 142 141 140 140 139 139 140 140 142 144 153 168 179 185
    185 182]]]]
```

上面的代码在每个时间点上计算所有体元的最大值。

```
stddev = series.map(lambda s: s.std())
print stddev.values._rdd.take(3)
print stddev.shape
...
[((0, 0, 0), array([ 0.4])), ((0, 0, 1), array([ 0.35707142]))]
(2, 76, 87, 20)
```

上面的代码计算每个时间序列的标准差并且返回结果 RDD，RDD 中保留了所有键。

也可以在本地将 Series 重新包装为对应特定形状的 NumPy 数组（这里为 2×76×87）：

```
repacked = stddev.toarray()
plt.imshow(repacked[:,:,0], interpolation='nearest', cmap='gray',
    aspect='equal')
print type(repacked)
print repacked.shape
...
<type 'numpy.ndarray'>
(2, 76, 87)
```

这时我们就可以用同样的空间关系绘制每个体元的标准差（如图 11-4 所示）。应该注意，不要向客户端返回太多数据，因为这样会占用很多带宽和内存资源。

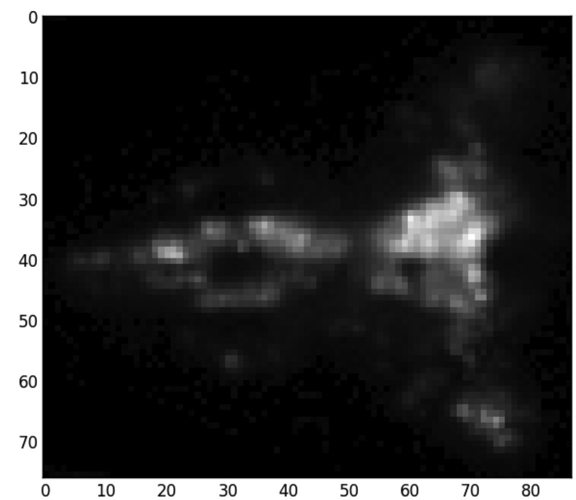


图 11-4：原始斑马鱼数据中每个体元的标准差

同样，可以通过绘制中部的时间序列来直接观察一下这些时间序列（如图 11-5 所示）：

```
plt.plot(series.center().sample(50).toarray().T)
```

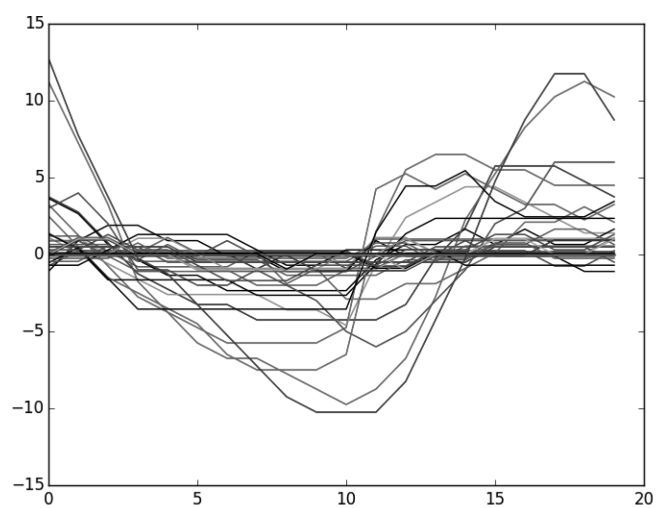


图 11-5：中部的时间序列的 50 个随机样本子集

在每个序列上应用任何用户定义函数也非常容易。只需使用 `map` 方法，它调用底层 RDD 的 `.map()` 方法，对底层键 – 值对的值进行处理。

```
series.map(lambda x: x.argmin())
```

Thunder核心数据类型

更一般地说，Thunder 的两个核心数据类型 `Series` 和 `Images` 都继承自 `Data` 类型，该类型最终包含由本地 NumPy 数组或 Spark RDD 支持的 `BoltArray`。`Data` 类代表键 – 值对的 RDD，键是语义标识符（比如空间坐标元组），值是一个由实际值组成的 NumPy 数组。比如，对 `Images` 对象而言，键可以是一个时间点，值是以 NumPy 格式数组存放的该时间点的图像。对 `Series` 对象而言，键可以是一个相应体元坐标的 n 维元组，值是表示该体元时间序列度量的一维 NumPy 数组。`Series` 中所有数组的维度必须相同。

通常，同样的数据集既可表示为 `Images` 对象也可表示为 `Series` 对象，这两个对象之间可以通过 shuffle 操作（代价可能非常高）进行相互转换（跟行式与列式表示相互转换类似）。

Thunder 的 `Data` 可以持久化为一组图像，按图像文件名的字母序排序，也可以持久化为一组 `Series` 对象的二元一维数组。要了解更多细节，请参考文档（http://docs.thunder-project.org）。

11.4　用Thunder对神经元进行分类

在本节示例中，我们将使用 K 均值算法对不同的斑马鱼时间序列进行聚类。聚类之后，这些时间序列将变成几个大类，用以描述不同类型的神经行为。我们将使用 GitHub 资料库上存储的 `Series` 数据，该数据比之前我们使用的图像数据要大。但是这些数据的空间分辨率很低，不足以区分神经元个体。

首先我们来加载数据：

```
# 这个数据集可以在ass库中下载
images = td.images.frombinary(
    '/user/ds/neuro/fish-long', order='F', engine=sc)
series = images.toseries()
print series.shape
...
(76, 87, 2, 240)
[  0   1   2   3   4   5   6   ...   234 235 236 237 238 239]
```

结果表明图像的维度和之前一样，但时间点由 20 个变成了 240 个。为了得到最好的聚类结果，我们要对特征进行归一化。

```
normalized = series.normalize(method='mean')
```

现在我们绘制一些时间序列图来看看这些时间序列的情况。使用 Thunder 可以在 RDD 上随机采样并按一定标准（比如默认的最小标准差）对集合元素进行过滤。为了选择一个合适的阈值，首先来计算每个时间序列的 stddev，然后对 10% 的样本值绘制直方图（如图 11-6 所示）：

```
stddevs = (normalized
    .map(lambda s: s.std())
```

```
        .sample(1000))
plt.hist(stddevs.values, bins=20)
```

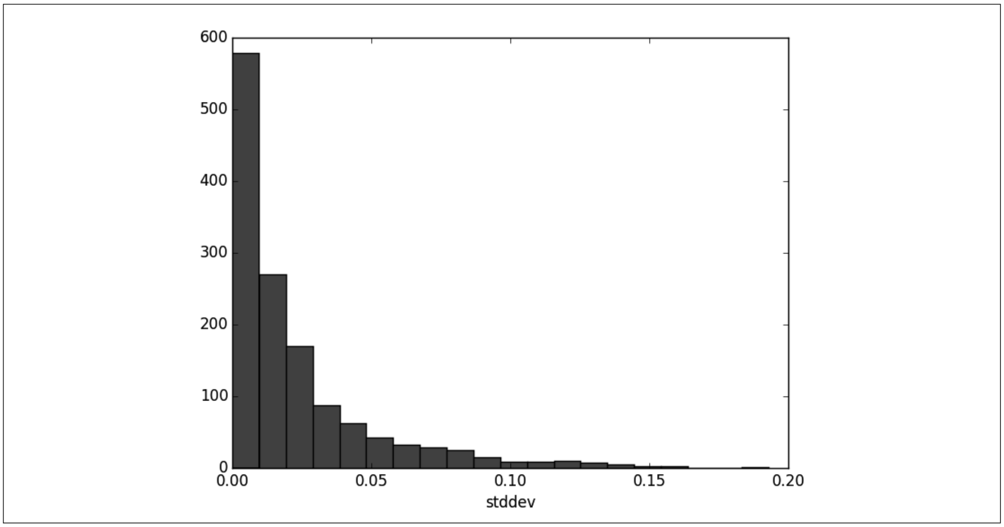


图 11-6：体元标准差分布

知道了标准差，我们选择阈值为 0.1，以便能得到大部分“活跃”的序列（见图 11-7）：

```
plt.plot(
    normalized
        .filter(lambda s: s.std() >= 0.1)
        .sample(50)
        .values.T)
```

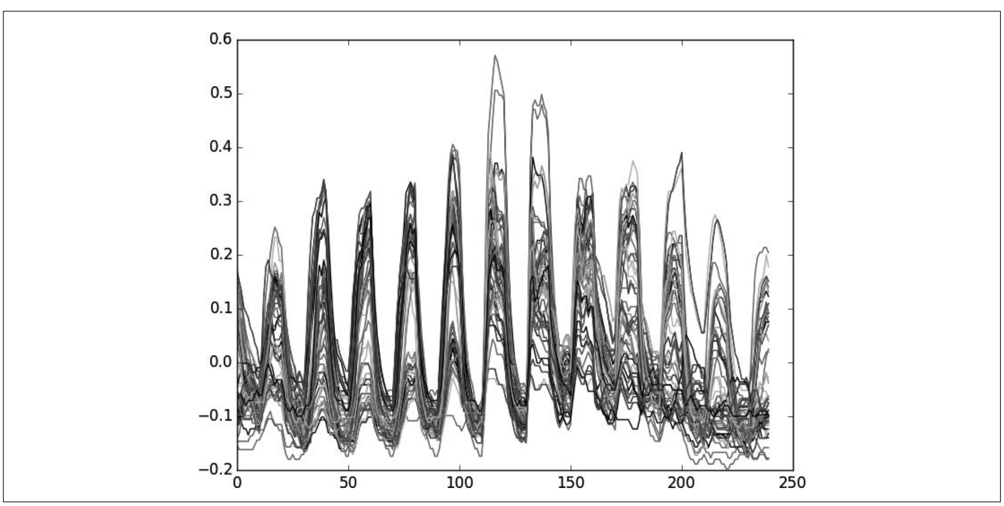


图 11-7：基于标准差的 50 个最活跃的时间序列

现在我们对数据有了一定了解，最后来将体元聚类成不同行为模式，使用 MLlib 的 K 均值功能。下面对多个 *k* 值运行 K 均值聚类：

```
from pyspark.mllib.clustering import KMeans
ks = [5, 10, 15, 20, 30, 50, 100, 200]
models = []
for k in ks:
    models.append(KMeans.train(normalized.values._rdd.values(), k))
```

现在对每个簇计算两个简单的误差指标。第一个指标简单地对时间序列到簇中心的欧氏距离求和。第二个指标是 `KMeansModel` 对象的一个内置指标：

```
def model_error_1(model):
    def series_error(series):
        cluster_id = model.predict(series)
        center = model.centers[cluster_id]
        diff = center - series
        return diff.dot(diff) ** 0.5

    return (normalized
        .map(series_error)
        .toarray()
        .sum())

def model_error_2(model):
    return model.computeCost(normalized.values._rdd.values())
```

我们对每个 *k* 值都计算这两个指标，并将结果绘制成图 11-8：

```
import numpy as np
errors_1 = np.asarray(map(model_error_1, models))
errors_2 = np.asarray(map(model_error_2, models))
plt.plot(
    ks, errors_1 / errors_1.sum(), 'k-o',
    ks, errors_2 / errors_2.sum(), 'b:v')
```

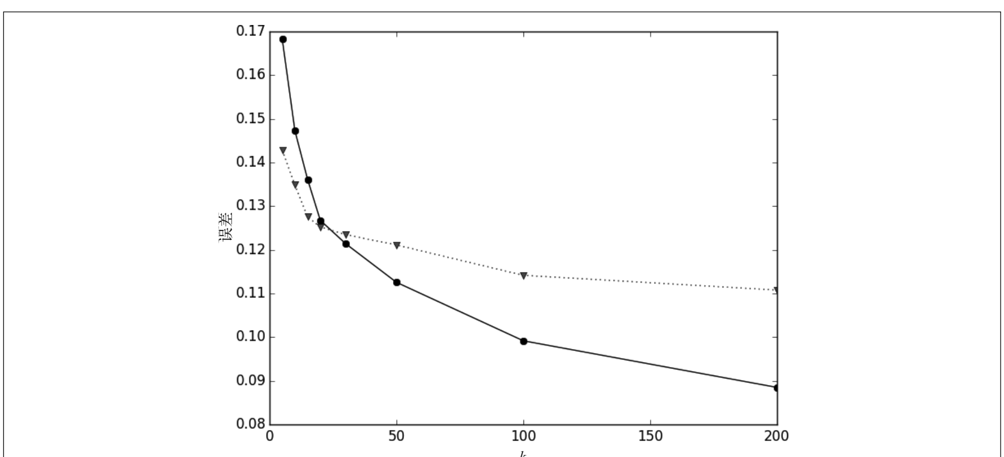


图 11-8：以 *k* 为变量的 K 均值误差指标函数（圆点代表 `model_error_1`，三角形代表 `model_error_2`）

我们预计这些指标通常是 k 的单调函数；曲线看起来在 k=20 处有一个明显的拐点。现在我们把从数据中学习得到的类簇中心画出来，如图 11-9 所示：

```
model20 = models[3]
plt.plot(np.asarray(model20.centers).T)
```

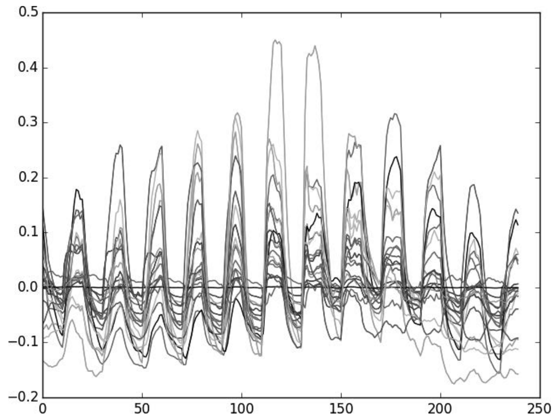


图 11-9：k=20 时的模型中心

给不同类簇的体元分配不同颜色并绘制图像本身也很简单，结果如图 11-10 所示：

```
import seaborn as sns
from matplotlib.colors import ListedColormap
cmap_cat = ListedColormap(sns.color_palette("hls", 10), name='from_list')
by_cluster = normalized.map(lambda s: model20.predict(s)).toarray()
plt.imshow(by_cluster[:, :, 0], interpolation='nearest',
    aspect='equal', cmap='gray')
```

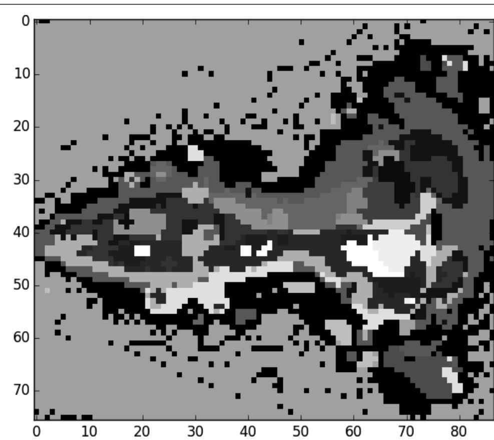


图 11-10：不同类簇的体元分配不同颜色的三维像素图

从图中显然可以看出，学习模型得到的聚类一定程度上刻画了斑马鱼大脑的解剖结构。如果原始数据分辨率高到足以看清亚细胞的结构，我们就可以先对体元进行 K 均值聚类，其中 *k* 等于图像数据中神经元的估计个数。然后，再定义每个神经元的时间序列，而这些时间序列可再用来聚类以确定神经元的不同功能类型。

11.5 小结

Thunder 项目还比较年轻，但功能已经比较丰富。除了时间序列统计和聚类，它还包含矩阵分解、回归 / 分类和可视化模块。Thunder 项目的文档和教程写得非常好，涵盖的功能也很多。如果想了解 Thunder 的使用，可以看一下 Thunder 作者在 2014 年 7 月的 *Nature Methods* 杂志上发表的文章“Mapping brain activity at scale with cluster computing”（http://www.nature.com/nmeth/journal/v11/n9/abs/nmeth.3041.html）。

作者介绍

桑迪·里扎（Sandy Ryza），目前在 Remix 公司从事公共交通算法方面的研发。在此之前，他是 Cloudera 公司和 Clover Health 公司的高级数据科学家。他是 Apache Spark 代码提交者、Apache Hadoop 项目管理委员会委员，以及 Time Series for Spark 项目的创始人。他曾获得布朗大学计算机科学系 2012 年“无厘头杯”（Twining award）最佳惊悚效果奖。

于里·莱瑟森（Uri Laserson），西奈山伊坎医学院的遗传学助理教授，他利用 Hadoop 生态系统开发了可扩展的基因组学和免疫学技术。

肖恩·欧文（Sean Owen），Cloudera 的数据科学总监。他是 Apache Spark 的代码提交者和项目管理委员会委员，也曾是 Apache Mahout 的代码提交者。

乔希·威尔斯（Josh Wills），Slack 公司的数据工程主管，同时也是 Apache Crunch 项目的创始人，曾经写过关于数据科学家的推文。

封面介绍

本书封面上的动物游隼是世界上最常见的掠食鸟类之一，地球上除了南极洲之外都可以见到它的身影。游隼的栖息地非常广泛，包括城市、热带、沙漠和苔原。有些游隼会从越冬地迁徙很长的距离到达夏季栖息地。

游隼是世界上飞行速度最快的鸟类，其俯冲速度达到每小时 320 公里。游隼的食物是其他鸟类，比如鸣鸟和野鸭，同时也吃蝙蝠，它们可以在半空中抓住猎物。

成年游隼的翅膀为蓝灰色，背部为黑褐色，腹部为米色且带有褐色斑点，脸部为白色，面颊上有黑色条纹。游隼的鸟嘴呈钩形，并且有一双有力的爪子。游隼的名字来自拉丁语“peregrinus”，意思是“盘旋”。游隼深受放鹰者的喜爱，数百年来都出现在放鹰运动中。

O’Reilly 用在封面上的很多动物都是濒危物种，它们全都对这个世界很重要。如果你想帮助它们，请访问 animals.oreilly.com。

封面图片源自 Lydekker 所著的 *Royal Natural History*。

微信连接

回复"Spark"查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230

图灵读者官方群II：164939616

图灵社区

iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Spark高级数据分析（第2版）

作为计算框架，Spark速度快，开发简单，能同时兼顾批处理和实时数据分析，因此很快被广大企业级用户所采纳，并随着近年人工智能的崛起而成为分析和挖掘大数据的重要得力工具。

本书由业内知名数据科学家执笔，通过丰富的示例展示了如何结合Spark、统计方法和真实世界数据集来解决数据分析问题，既涉及模型的构建和评价，也涵盖数据清洗、数据预处理和数据探索，并描述了如何将结果变为生产应用，是运用Apache Spark进行大数据分析和处理的实战宝典。

第2版根据新版Spark最佳实践，对样例代码和所用资料做了大量更新。

本书涵盖模式如下：

- **音乐推荐和Audioscrobbler数据集**
- **用决策树算法预测森林植被**
- **基于K均值聚类进行网络流量异常检测**
- **基于潜在语义算法分析维基百科**
- **用GraphX分析伴生网络**
- **对纽约出租车轨迹进行空间和时间数据分析**
- **通过蒙特卡罗模拟来评估金融风险**
- **基因数据分析和BDG项目**
- **用PySpark和Thunder分析神经图像数据**

“本书是大数据市场翘楚Cloudera公司经验总结，通过案例分析详尽展现了解决问题的全过程，自第1版出版后一直位列亚马逊网站大数据分析类图书前茅。中文版的问世，实在是国内技术圈的幸事。”

——苗凯翔

思科中国研发公司CTO

前Cloudera公司副总裁

桑迪·里扎（Sandy Ryza），Spark项目代码提交者、Hadoop项目管理委员会委员，Time Series for Spark项目创始人。曾任Cloudera公司高级数据科学家，现就职于Remix公司从事公共交通算法开发。

于里·莱瑟森（Uri Laserson），MIT博士毕业，致力于用技术解决遗传学问题，曾利用Hadoop生态系统开发了可扩展的基因组学和免疫学技术。目前是西奈山伊坎医学院遗传学助理教授，曾任Cloudera公司核心数据科学家。

肖恩·欧文（Sean Owen），Spark、Mahout项目代码提交者，Spark项目管理委员会委员。现任Cloudera公司数据科学总监。

乔希·威尔斯（Josh Wills），Crunch项目发起人，现任Slack公司数据工程主管。曾任Cloudera公司高级数据科学总监。

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议　计算机 / 大数据 / Spark

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-48252-5

9 787115 482525 >

ISBN 978-7-115-48252-5

定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks